



الترجمات



مراحل المترجم

الاسم: ٢٥/٩/٢٠٠٦

مقدمة:

– المترجم compiler هو عبارة عن أداة لتحويل



نص مكتوب بلغة ما source file إلى شيء

قابل للتنفيذ على الحاسب executable أو إلى

نص مكتوب بلغة أخرى

– غالباً لا نضطر لأن نكتب مترجم كامل ولكننا نكون بحاجة لبعض أجزائه فقط كما هو الحال عندما

نصمم برنامج رسم يكون دخله ملف يحتوي على عبارات من الشكل:

Circle x, y, c

Rectangle x1, y1, x2, y2

هنا عندما نريد أن نفتح ملف ما لنرسمه بهذا البرنامج فإننا نحتاج للتأكد من تنسيق هذا الملف حيث

يجب أن يحتوي على كلمة circle متبوعة بثلاثة قيم أو rectangle متبوعة بأربعة قيم ولا يجب أن

يحتوي على أي شيء آخر أبداً

عادة نقوم بعملية التحقق القواعدي هذه حتى نتأكد من أن هذا الملف قابل للرسم من قبل برنامجنا فقد

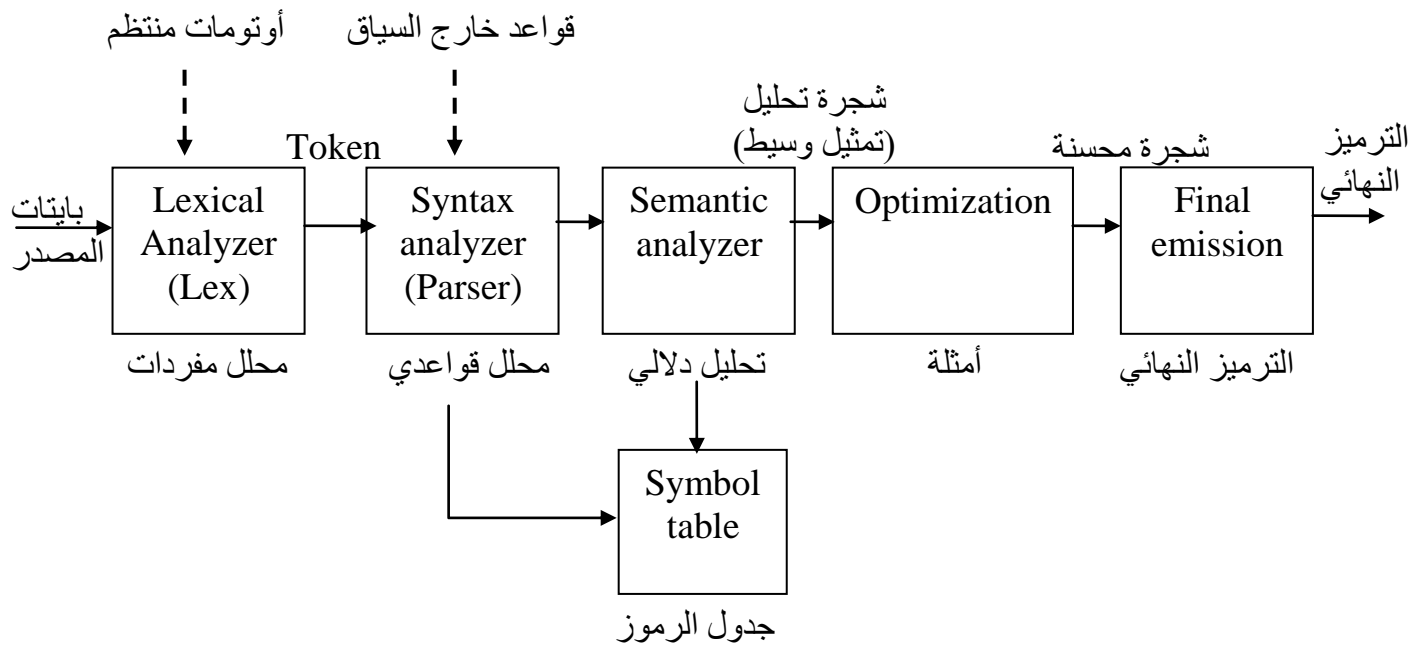
يكون الملف مجتزأ لأنه معطوب أو أن أحد الأشخاص قد قام بالتعديل بشكل خاطئ عليه

ملاحظة: يعطي المفسر النتائج فوراً أي أنه ينفذ التعليمات بدون مرحلة توليد executable بعكس

المترجم

مراحل المترجم:

هناك العديد من المراحل التي لابد من المرور بها حتى نستطيع الوصول إلى الملف التنفيذي انطلاقاً من ملف نصي، وهذا ما يبينه المخطط التالي: (سنهتم بتفاصيل المراحل الأولى من المترجم بينما لن نهتم بالتفاصيل في المراحل الأخيرة منه لذلك فهي غير موجودة بالتفصيل في المخطط)



ملاحظة: أحياناً للتبسيط نقوم بحذف بعض من المراحل السابقة كما هو الحال في الحواسيب القديمة التي كانت تحتوي على زر خاص بكل token حتى لا نضطر لاستعمال محلل مفردات، أي أننا كنا نضغط زر واحد لكتابة كلمة if على الشاشة وتوليد token الخاص بها مباشرة ووضعها في الذاكرة لنشرح الآن بشكل سريع ومبسط مراحل المترجم الموجودة في المخطط السابق:

محلل المفردات:

– يقوم بتجميع عدة حروف من ملف المصدر معاً ليشكل منها كائن واحد هو token

– Token: هو كل مفردة مستقلة فقد يكون حرف واحد أو عدة حروف ويمثل بـ struct مكون من حقلين هما:

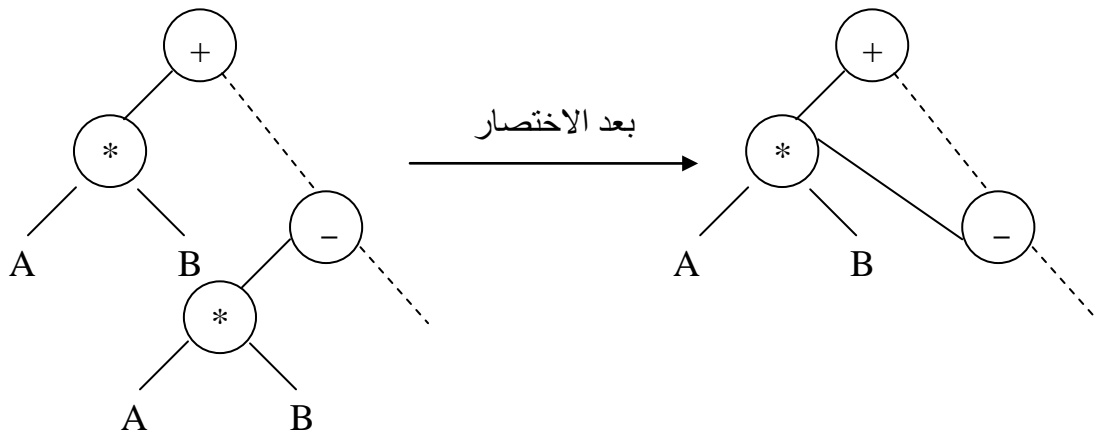
- النوع: ويحدد نوع token مثل أن يكون integer أو if

- القيمة: ويحدد قيمة ال token في حال وجودها مثل 220
- فمثلاً كلمة if هي token وكذلك for وكذلك + وكذلك ++ وكذلك (و...)
- المحلل القواعدي:

- يقوم بالتأكد من مطابقة tokens لقواعد اللغة المعرفة مسبقاً
- وينتج عنه شجرة تحليل تشبه إلى حد كبير الشجرة التي تنتج من توليد قواعد خارج السياق لتعبير ينتمي للغة الخاصة بهذه القواعد
- التحليل الدلالي:

- نقوم هنا بالتأكد من تطابق الأنماط بين التعبيرات والمتحولات المسندة لها
- وكذلك نتأكد من عدم استعمال متحول غير معرف مسبقاً
- نتعامل هذه المرحلة بشكل رئيسي مع جدول الرموز symbol table
- الأمثلة:

- هنا نقوم باستبدال الأشياء المكررة في هذه الشجرة بـ reference عليها وذلك لتحسين الأداء حيث أننا نوفر حساب نفس القيمة لأكثر من مرة:



- لاحظ أننا اختزلنا الجداء المتكرر حيث جعلنا الابن اليساري للطرح هو ناتج الجداء المحسوب سلفاً والذي لا داعي لحسابه من جديد

ملاحظة:

عادة لا تمر المفسرات بهذه المرحلة لأنها تنفذ فوراً (إلا إذا أردنا إضافة مثل هذا التحسين من أجل اختصار التنفيذ المتكرر للأشجار الكبيرة جداً وهذا أمر نادر جداً)

الترميز النهائي:

– هنا يتم توليد الترميز النهائي والذي قد يكون assembly أو نص text مكتوب بلغة أخرى تختلف عن لغة مصدر المترجم

ملاحظة:

هناك بعض المراحل التي تخص الترميز النهائي والتي لم نوردتها في الجدول بسبب عدم اهتمامنا بها في مقرر المترجمات لأنها تهتم بآليات كتابة assembly وإنشاء الملف التنفيذي exe

المفسر Interpreter:

– يشترك المفسر مع المترجم بأغلب المراحل مع بعض الاختلافات البسيطة

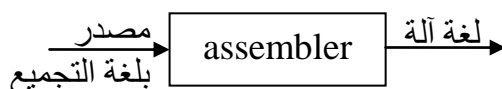
– في الغالب لا يكون في المفسر تحسينات ولا مرحلة توليد code لأنه ينفذ كل تعليمة في حال اكتمالها (لاحظ لغة HTML أو PHP)

– كما أنه غالباً لا يوجد أنماط للمتحويلات في جدول الرموز لأن اللغة تكون مرنة جداً بالتحويل بين الأنماط إذا اضطر الأمر، كما أنها تسمح باستخدام المتحويلات مباشرة بدون تعريف (تذكر لغة (PHP

– بسبب ذلك (عدم وجود أنماط للمتحويلات) لا يوجد غالباً تحليل دلالي في المفسر (يوجد شيء قليل من التحليل الدلالي فقط)

نظرة تاريخية:

لغات assembly:



– يكون لدينا هنا assembler

– غالباً ما يكون لدينا header في بداية ملف التشغيل

exe مثل حجم الذاكرة المطلوبة لهذا البرنامج أو حجم stack ومعلومات أخرى خاصة بالملف التنفيذي...

لغات ذات هيكلية محددة formatted:

– مثل لغة فورتران fortran حيث لدينا في كل سطر شكل محدد يجب على المبرمج إتباعه (أول 5 bytes تمثل رقم السطر ثم البايت السادس يكون فارغ أو يحتوي على الحرف c الذي يدل على أن هذا السطر عبارة عن تعليق comment ثم لدينا 70 محرف للتعليمة)

– وكانت هذه الهيكلية تسبب الكثير من الصعوبات للمبرمج

لغات غير مهيكلة unstructured:

– هنا لا يوجد مفهوم الكتلة block

– مثل basic و fortran

لغات مهيكلة structured:

– هنا يوجد لدينا مفهوم الكتلة ومفهوم scope

– كما في c و pascal حيث لدينا كتلة (حلقات) ولكن لا نستطيع تعريف محتويات داخل هذه الحلقات

لغات غرضية التوجه:

– هنا يوجد لدينا مفهوم الصف

– كما في لغة C++

– في اللغات غرضية التوجه يقوم المترجم compiler بعدة مسحات على النص (على الأقل مرتين) ففي المرور الأول على النص يقوم بالتعرف على الـ classes و methods الخاصة بكل class (أي class structure) ولا يهتم بصحة التعليمات أبداً، ثم وفي المرور الثاني يعالج صحة التعليمات مما يجعله يتخلص من مشكلة استخدام الـ class قبل تعريفه أو استخدام class1 لـ class2 واستخدام class2 لـ class1

اللغات التي تعمل على آلة وهمية virtual machine:

– مثل لغة java أو C# (لغات .NET. بشكل عام)

– هنا نمر بكل مراحل compiler ولكننا في النهاية نحتاج لمفسر حتى يفسر byte code لأنه ينفذ فوراً كما أننا نحتفظ هنا بما يتم تحويله حتى إذا ورد مرة ثانية فإننا نقوم باستدعائه فقط

مفاهيم safe code و garbage collection

– والتي نأمل أن نتطرق لها في نهاية الفصل إذا اتسع الوقت لذلك
أدوات توليد المترجمات:

– لدينا العديد من الأدوات لإنشاء المترجمات وفق اللغة التي سيكتب المترجم بها

:Java

– لدينا javacc (java compiler compiler) والذي سمي بهذا الاسم لأنه عبارة عن compiler يولد compiler بلغة java

– يكون ملف الدخل بلاهقة .cc وملف الخرج هو .java. ويحتوي على compiler

:C

– لدينا Lex على نظام UNIX (flex على MS-DOS) الذي يقوم بتوليد محلل مفردات مكتوب بلغة C (وليس C++ أي أنه لا يحتوي على classes)

– يحتوي الملف الناتج على التابع yylex والذي يعطي عند كل استدعاء له token جديد من ملف الدخل وينتظر الاستدعاء التالي

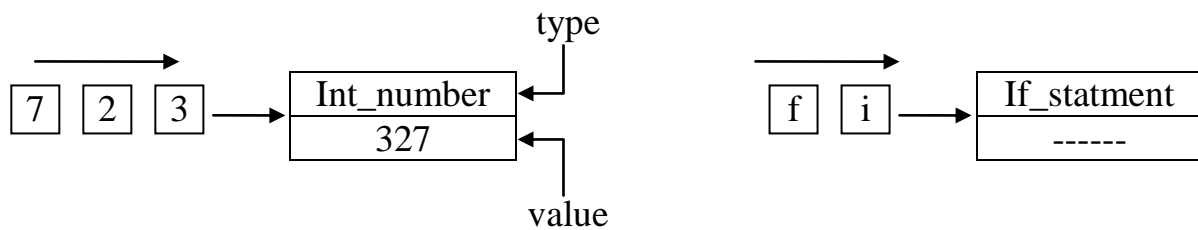
– لدينا Yacc (yet another compiler compiler) على نظام UNIX (bison) على MS-DOS الذي يولد محلل قواعدي مكتوب بلغة C

محلل المفردات Lexical Analyzer:

أهم الأعمال التي يقوم بها محلل المفردات:

١. تجميع محارف الدخل في مفردات tokens

حيث يجمع خانات الرقم لتوليد integer مثلاً كما يجمع حروف الكلمات المحجوزة keywords



- إن نوع المفردة token type هنا هو ثابت من النوع integer (final int) وليس string
 - يمكننا أن نعرف token ذو نمط statement وتكون قيمته هي If أو for كما يمكننا أن نعرف token خاص بكل نوع من أنواع statement كما في المثال السابق الذي يحتوي على token خاص بـ if

- إن الطريقة الثانية (تعريف token لكل نوع من أنواع statement) يسبب زيادة عدد المفردات ولكنه يوفر علينا الكثير عند كتابة القواعد حيث يجنبنا التأكد من نوع statement في كل مرحلة من كل قاعدة

مثال:

لنكتب قاعدة If البسيطة (بدون else) في كلتا الحالتين:

تعريف token خاص بـ If:

<if> if_statement <exp> then_kw <statement>

تعريف token واحد لل statement ككل:

يجب أن نحذف if_statement ونضع بدل منها <statement> ثم نضع action ليتحقق من نوعها هل هي if أم لا

٢. التخلص من التعليقات comments والمحارف الزائدة (أسطر فارغة أو space أو tab)

يتم التخلص من هذه المحارف الزائدة في هذه المرحلة أي قبل الوصول لقواعد اللغة وذلك حتى لا نعقد القواعد فنكتب:

<if> → if_statement(' ' | '\n') ...

لاحظ أننا سنكون مضطرين لإضافة المحارف الزائدة في العديد من الأماكن ضمن نفس القاعدة وهذا مريب جداً كما أنه سيجعل القاعدة غير مفهومة أبداً

لذلك يكون من الأبسط جعل محلل القواعد لا يرى المحارف الزائدة أبداً

٣. إدارة أرقام الأسطر والأعمدة لكل token

وذلك من أجل إعادة رقم السطر والعمود عند حصول الأخطاء

لاحظ أننا نكون مضطرين لحفظ أرقام الأسطر والأعمدة لأننا في الكثير من الأحيان لا نعرف بوجود الخطأ إلا بعد تجاوز مكانه فمثلاً في حالة قاعدة المساواة:

$\langle \text{assignment} \rangle \rightarrow \text{id assign} \langle \text{exp} \rangle \text{semicolon} \{ \text{action} \}$

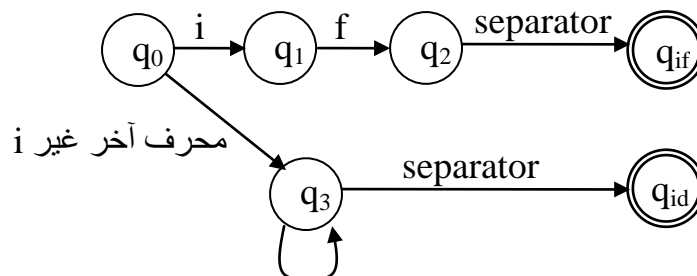
قد يظهر خطأ في action حيث يكون لدينا عدم تطابق في الأنماط بين طرفي المساواة، وهذا الخطأ سيتم كشفه في action أي بعد الفاصلة المنقوطة والتي قد تكون بعد عشرة أسطر من المساواة، لذلك فإنه يتوجب علينا حفظ مكان كل token (سطر وعمود)

لذلك على الأغلب يكون مع كل token رقم سطر ورقم عمود

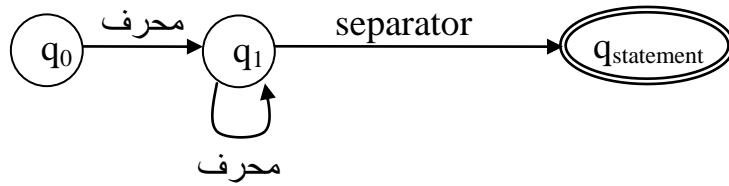
٤. من مهام محلل المفردات تصغير الأحرف في اللغات غير الحساسة للأحرف مثل pascal بينما لا يوجد شيء مشابه لذلك في اللغات الحساسة للأحرف

مثال عام:

لنرسم الأوتومات الذي يمثل المحلل المفرداتي لـ compiler للغة لا تحتوي إلا على If و id



لاحظ تعقيد الأوتومات على حساب القواعد، حيث كان بإمكاننا ببساطة أن نكتب:



وبذلك نجعل المحلل القواعدي يتأكد كل مرة من نوع token المولد من قبل هذا الأوتومات (هل هو If أم id أم ...)

مثال:

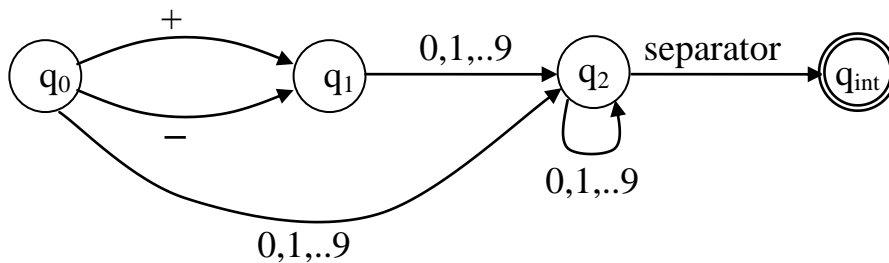
لنكتب الأوتومات الذي يتعرف على العدد الصحيح

إن العدد الصحيح يبدأ بـ + أو - ثم رقم عشري ويليه أرقام عشرية

كما يمكننا أن نضيف إذا أردنا مفهوم الأعداد المكتوبة بالصيغة العلمية مثل 2E3 أي 2×10^3

كما ينتهي العدد الصحيح بـ separator (فراغ أو tab أو enter أو + أو ...)

ملاحظة: يكون separator نهاية لـ token وبداية لآخر أو يكون زائد بلا فائدة



إن كل حالة غير موجودة في الأوتومات تؤدي إلى خطأ

كما نعلم فإن الأوتومات الحتمي المنتهي يكافئ التعبير المنتظم لذلك فإنه يمكننا كتابة التعبير المنتظم التالي المكافئ للأوتومات السابق:

Id_number: $(+ | - | \epsilon) (0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9)^+$

ملاحظة أخيرة: ذكر الدكتور أن مرجع المقرر هو:



Modern Compiler Implementation in Java, Second Edition

by Andrew W. Appel and Jens Palsberg

ISBN:052182060x

That 's all folks



lectures_team@hotmail.com

المحلل القواعدي

الاسم: ٢/١٠/٢٠٠٦

تذكرة:

– عددنا في المحاضرة السابقة مراحل المترجم وقلنا أن المرحلة الأولى هي تحليل المفردات ثم يأتي المحلل القواعدي

– ولكن في بعض الأحيان كما في لغة C++ يكون لدينا مرحلة سابقة لمحلل المفردات هي preprocessor والتي تتمثل بالأسطر التي تبدأ بالإشارة # فمثلاً يمكننا أن نكتب:

#define PI 3.14

#include <iostream>

– هنا لا يكون للمترجم أي علاقة بهذه الأسطر لأن preprocessor يقوم بفتح الملف بالكامل واستبدال كل PI بـ 3.14 لذلك فإن المترجم لا يرى PI أبداً كما أن preprocessor يقوم بفتح ملف iostream وإحضاره للمترجم

ملاحظة:

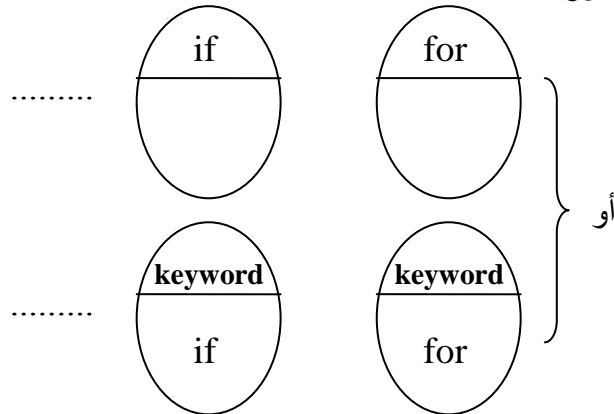
تتم عملية فتح الملفات الموضوعة بعد #include بشكل عودي أي أن كل ملف يتم فتحه يتم الدخول إليه وفتح كل الملفات الموجودة بعد كلمة #include فيه وهكذا... (وهذا ما كان يسبب الكثير من المشاكل عندما يقوم file1 بـ include لـ file2 ويقوم file2 بـ include لـ file3)

– إن محلل المفردات كما نعلم عبارة عن أوتومات منتظم وقبل أن نقوم بكتابته يجب علينا أن نعرف مفردات هذه اللغة (أي tokens الخاصة بها)

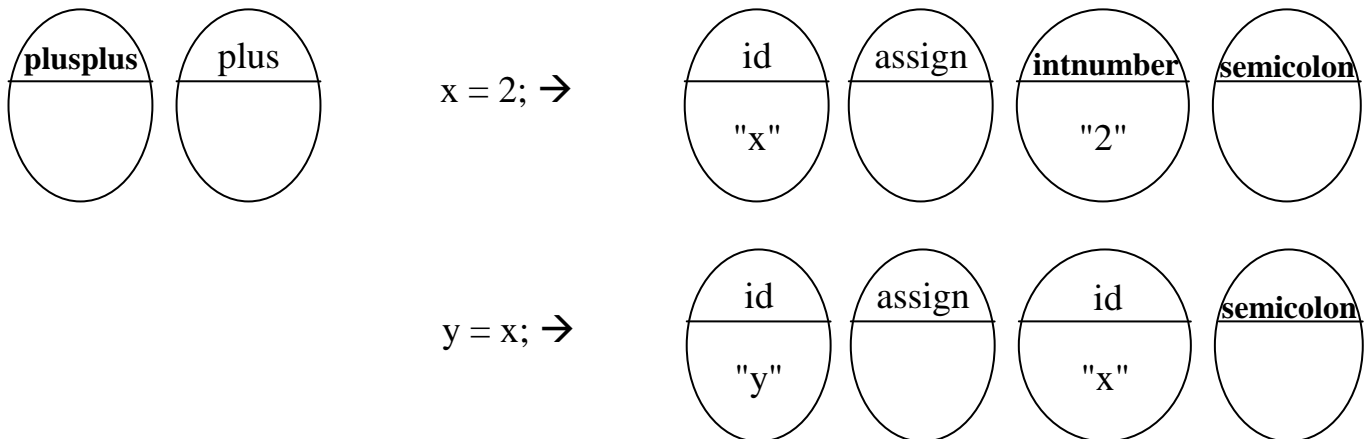
مثال:

لنحدد بعض tokens الخاصة بلغة Java:

الكلمات المحجوزة:



بعض العبارات:

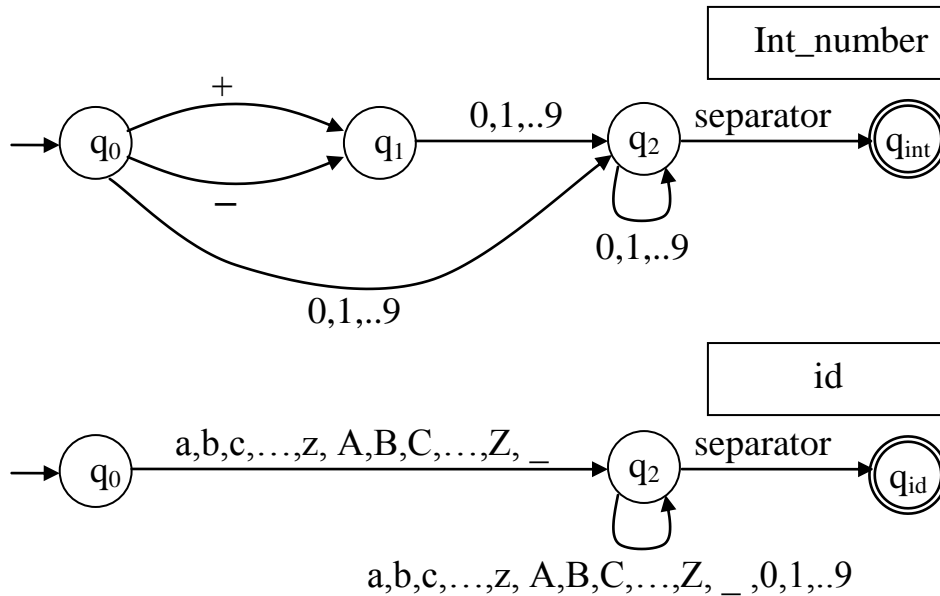


لاحظ أن لدينا token هو $(plus)+$ ولدينا token هو $(plusplus)++$ أي أن لهما نفس البداية، لذلك فإن محلل المفردات يحاول المطابقة مع أكبر token ممكن المطابقة معه حتى يستطيع في حالة $++$ أن يعطي token واحد هو $plusplus$ لا أن يعطي $plus$ token ثم $plus$ token

مثال:كتابة محلل مفردات يتعرف على `id` و `int_number`

هنا لدينا 2 tokens فقط

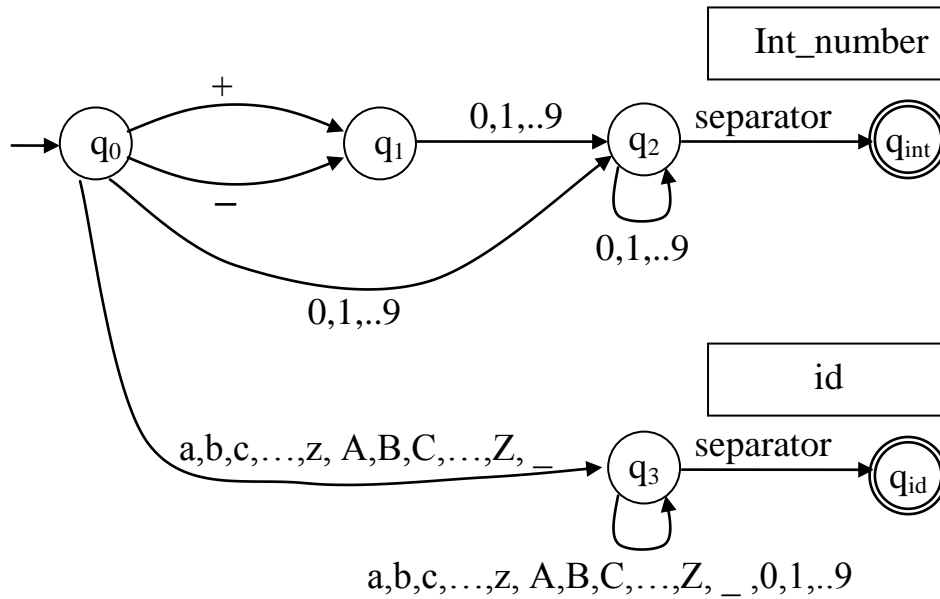
سنكتب أوتومات لكل token على حدا ثم نقوم بدمج الأوتوماتين معاً للحصول على الأوتومات الخاص بمحلل المفردات



في كل خطوة (حالة من حالات الأوتومات) نقوم بتجميع string ليحتوي في النهاية على قيمة token الذي سيتم توليده لاحقاً

ملاحظة: قد نحتاج في بعض الحالات إلى إعادة ال separator (push back) ليتم أخذه مرة ثانية كما لو كان separator هو + حيث نكون بحاجة لإعادته مرة ثانية بعد أن قرأناه حتى نقرأه مرة ثانية ونقوم بإنشاء token خاصة به. يتم ذلك عملياً إما باستخدام unget في حال وجودها كإجرائية جاهزة في stream الذي يتم استخدامه لقراءة ملف المصدر، أو عن بواسطة طرق خاصة بمصمم المترجم مثل أن يقوم بالاحتفاظ في كل مرة بالحرف السابق للحرف الحالي من الملف ومعالجة هذا الحرف كحرف أساسي بينما اعتبار الحرف الحالي في الملف هو نظرة للأمام فقط look ahead

والآن لندمج الأوتوماتين السابقين معاً (يفضل دائماً أن نقوم بدمج الأوتوماتات باليد لا عن طريق إجراءات جاهزة لأنها عادة ما تسبب إنشاء وصلات ε التي تسبب الكثير من الإرباك عند تطبيق الأوتومات على الحاسب)



لدينا طريقتين لتحويل هذا الأوتومات إلى محلل مفرداتي:

◀ مصفوفة ثنائية: هنا يكون بعدا المصفوفة هما الحالة الحالية والدخل الجديد وتعطينا المصفوفة

الحالة الجديدة التي يجب أن ننتقل إليها (وهذا مشابه تماماً لتابع الانتقال في الأوتومات)

◀ إنشاء إجرائية خاصة بكل حالة: حيث تقوم هذه الإجرائية بالتحقق من الدخل واستدعاء الإجرائية

المناسبة (أي الإجرائية المقابلة للحالة التي يجب الذهاب إليها)

لنكتب مثلاً عن محلل مفردات باستخدام طريقة الإجرائيات:

class Token

{

private int type;

private String value;

static final int ID=1;

static final int INT_NUM=2;

static final int ERROR=3;

static final int EOF=4;

Token(){/*Some needed code*/}

void set(){/*Set the fields with values*/}

void get(){/*Get values form the fields*/}

}

public class lexical

```
{
    private String t; //To join what we have read until now
    private String filename; //Usually we need this field
    public lexical(String fn)
    {
        /*Code to handle: open the file in a stream*/
        t="";
        filename=fn;
    }
    private char get()
    {
        /*Code to return the next character of the file*/
    }
    private void unget()
    {
        /*Code to return the last character to the stream*/
    }
    public Token nextToken() //Return the next token to the parser
    {
        t="";
        return q0();
    }
    private Token q0()
    {
        char c=get();
        t=t+c;
        if((c=='+')||(c=='-')) return q1();
        if((c>='0')&&(c<='9')) return q2();
        if (/*c is an upper case or lower case or _ */) return q3();
        return new Token(Token.ERROR,"");
    }
}
```

```

private Token q3()
{
    char c=get();
    if(/*c is an upper case or lower case or _ or number*/)
    {
        t=t+c;
        return q3();
    }
    if(isSeparator(c))
    {
        unget();
        return qid();
    }
    return new Token(Token.ERROR,"");
}
private Token qid()
{
    return new Token(Token.ID,t);
}
}

```

ملاحظات:

- ◀ لقد قمنا بكتابة بعض إجراءات المثال السابق وليس كلها، كما تركنا بعض الأشياء الصغيرة مثل آلية التعامل مع الملفات حتى لا نغوص كثيراً في التفاصيل الزائدة عن الحاجة
- ◀ عادة يقوم محلل القواعد parser بإنشاء object من lexical class لذلك نلاحظ أن الـ constructor الخاص بـ lexical من النوع public بينما constructor الخاص بـ Token هو من النوع private لأن lex هو الوحيد الذي ينشئ objects منه
- ◀ غالباً ما يكون لدينا token خاص بنهاية الملف حتى يعرف parser بانتهاء class lexical من عمله
- ◀ لاحظ أن الإجراءات الوحيدة التي هي من نوع public هي البناء constructor الخاص بـ Lexical بالإضافة إلى إجراءية nextToken لأن parser لا يحتاج لأكثر من ذلك

◀ لاحظ أن إنشاء object من نوع token يتم فقط في الحالات النهائية من الأوتومات أو في حال حصول خطأ

◀ لاحظ كيف تمثلت الحالات الغير موجودة في الأوتومات بأخطاء عند تحقيق هذا الأوتومات ب class lexical

المحلل القواعدي **parser**:

– يتحقق المحلل القواعدي من صحة البرنامج المكتوب من الناحية القواعدية

– حيث يعبر عن قواعد البرنامج اللغوية باستعمال قواعد خارج السياق

سؤال:

لماذا لا نستخدم أوتومات منتظم للتعبير عن قواعد لغة البرمجة بدل قواعد خارج السياق؟

الجواب:

لأن لغة البرمجة لا يمكن التعبير عنها بأوتومات منتظم بسبب حاجتها إلى ذاكرة غير منتهية (إذا طبقت بالأوتومات المنتظم)، وكمثال على هذه الذاكرة يمكننا أخذ تطابق الأقواس الذي يحتاج لعدد لعد الأقواس المتداخلة ولا يمكننا القيام بذلك باستخدام أوتومات منتظم لأنه لا يملك ذاكرة إلا عن طريق حالاته (كل حالة تمثل شيء معين)، مما يعني أننا سنحتاج لذاكرة غير منتهية (عدد حالات غير منتهي) حتى نستطيع تمثيل لغة البرمجة وهذا غير ممكن عملياً. كما يمكننا نفي استخدام الأوتومات المنتظم بسبب عودية التعابير expression الممكن كتابتها في لغة البرمجة خاصة إذا كان لدينا تعبير ضخم ويحتوي على العديد من التعابير الجزئية في داخله بعض مشاكل استخدام القواعد خارج السياق:

الغموض ambiguity:

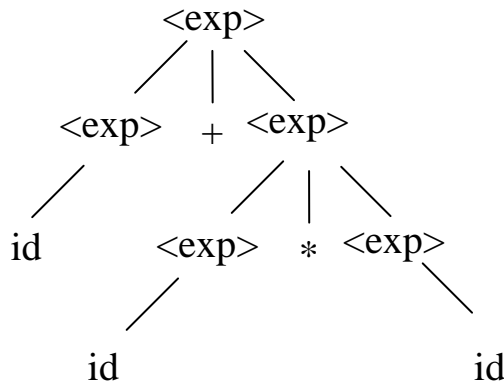
أي تحليل نفس الجملة بطريقتين مختلفتين صحيحتين

لذلك يتوجب علينا دائماً كتابة قواعد خارج السياق بدون غموض

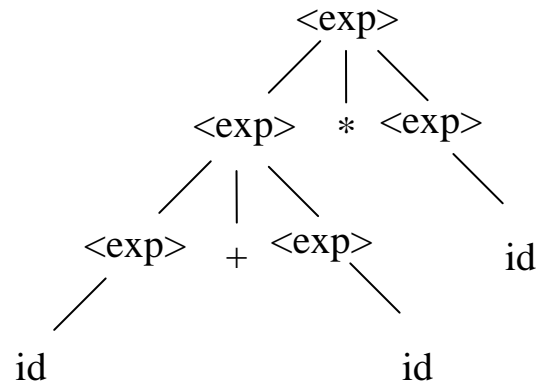
مثال:

$\langle \text{exp} \rangle \rightarrow \langle \text{exp} \rangle \pm \langle \text{exp} \rangle$
 $\quad \quad \quad | \quad \langle \text{exp} \rangle * \langle \text{exp} \rangle$
 $\quad \quad \quad | \quad \text{id}$

إن القواعد السابقة تحتوي على غموض لأن لدينا قاعدة يبدأ طرفها اليميني بنفس طرفها اليساري
 فلو حاولنا تحليل العبارة $a + b * c$ لوجدنا لها شجرتي تحليل صحيحتين هما:



أولوية للضرب



أولوية للجمع

هنا يمكن أن يتولد لدينا شجرتين ونحن (كبشر!!) غير قادرين على أن نقول أي منهما هي الصحيحة بدون الاستعانة بأولوية العمليات الحسابية

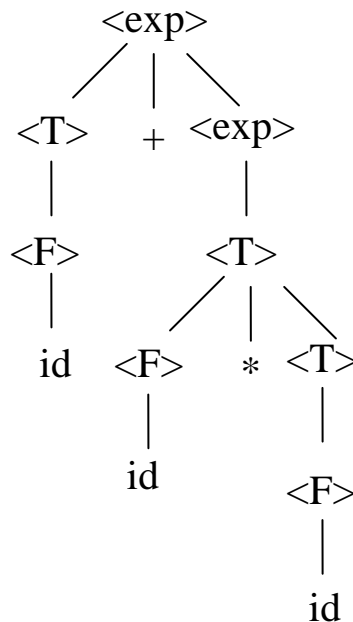
فلحل هذا الغموض لابد أن نستفيد من أولويات العمليات الحسابية فنكتب:

$\langle \text{exp} \rangle \rightarrow \langle T \rangle + \langle \text{exp} \rangle$
 $\quad \quad \quad | \quad \langle T \rangle$
 $\langle T \rangle \rightarrow \langle F \rangle * \langle T \rangle$
 $\quad \quad \quad | \quad \langle F \rangle$
 $\langle F \rangle \rightarrow \text{id}$

ملاحظات:

- ◀ لاحظ أن لدينا قاعدين لـ $\langle \text{exp} \rangle$ تبدأ بنفس البداية $\langle T \rangle$ وهذا لا يسبب مشاكل في بعض الأدوات المستخدمة للاستفادة من هذه القواعد
- ◀ في حل الغموض السابق طبقنا أولوية الضرب على الجمع، لاحظ أن الجمع يأتي في مستوى أعلى من الضرب لتحقيق هذه الأولوية (لاحظ ذلك من خلال شجرة التحليل وكذلك من خلال القواعد)
- ◀ إذاً كلما كانت العملية الحسابية ذات أولوية أعلى كلما كانت موجودة في مستوى أدنى من شجرة التحليل حتى تنفذ قبل غيرها من العمليات الحسابية
- ◀ وضعنا $\langle T \rangle$ كقاعدة من قواعد $\langle \text{exp} \rangle$ حتى نحل مشكلة التعبير الذي لا يحتوي على عمليات جمع نهائياً ولكنه يحتوي على عمليات ضرب والذي يعد تعبير حسابي
- بعد استخدام القواعد الخالية من الغموض السابقة أصبح بإمكاننا أن نرسم شجرة التحليل التالية للتعبير:

$$a + b * c$$



ملاحظة: عادة يعطي محلل القواعد parser أخطاء أكثر من محلل المفردات lex ولكن محلل المفردات قد يعطي أخطاء أيضاً كما هو الحال إذا صادف رمز غير موجود في كل اللغة (مثلاً في لغة

pascal الرمز % غير معرف أبداً فهو لا يمثل أي عملية حسابية كما أنه لا يمكن أن يرد ضمن اسم متحول أو ثابت)

ملاحظة: عادة عندما يحصل خطأ ما فإن compiler يفترض عدم وجود هذا الخطأ ويتابع بترجمة الملف ليعطي الأخطاء الأخرى (أي أنه يحاول إيجاد تصحيح لهذا الخطأ ويفترض أن هذا التصحيح موجود ثم يتابع بالترجمة) مما يجعل الكثير من الأخطاء تنتج عن خطأ وحيد إذا تم تصحيحه زالت جميع هذه الأخطاء (على الرغم من وجود compilers "ذكية" نسبياً والتي يكون تصحيحها للخطأ صحيحاً في كثير من الحالات كما هو حال compiler الخاص بـ java)

That 's all folks



lectures_team@hotmail.com



المتريجات



الاثنين ١٠/١٠/٢٠٠٦

المحلل القواعدي

أنواع التحليل القواعدي:

النوع الأول recursive-descent :

-تتم عملية بناء الشجرة هنا من الأعلى للأسفل

-نخصص لكل رمز غير نهائي تابع، وهذا شبيهة بالمحلل المفرداتي الذي كنا نخصص فيه لكل حالة من حالات الأوتومات تابع خاص بها

مثال:

-قمنا في المحاضرة السابقة بإنشاء lexical class الذي يملك constructor يأخذ اسم الملف كوسيط، ويحتوي على إجرائية nextToken

-انكتب الآن مثال لمحلل قواعدي للتعبير expressions:

$$\begin{aligned} \langle \text{exp} \rangle &\rightarrow \langle T \rangle \pm \langle \text{exp} \rangle \$ \\ &| \langle T \rangle \$ \\ \langle T \rangle &\rightarrow \langle F \rangle * \langle T \rangle \\ &| \langle F \rangle \\ \langle F \rangle &\rightarrow \underline{\text{id}} \\ &| \underline{\text{num}} \\ &| (\langle \text{exp} \rangle) \end{aligned}$$
ملاحظات:

◀ إن القواعد السابقة هي قواعد التعبير expression بعد إزالة الغموض وهو ما قمنا به في المحاضرة السابقة

◀ إن الرموز المسطرة هي رموز نهائية terminal يجب أن نجدها كما هي تماماً في الملف

◀ إن الأقواس الموجودة في قاعدة $\langle F \rangle$ تقع في أقصى عمق لقاعدة $\langle \text{exp} \rangle$ مما يعطيها أفضل أولوية على الإطلاق (راجع المحاضرة السابقة للتوسع بمفهوم العمق والأولوية)

◀ إن \$ الموجودة في نهاية قواعد $\langle \text{exp} \rangle$ تمثل eof وهي عبارة عن token يعيده محلل المفردات حتى يتعرف محلل القواعد على انتهاء الملف

◀ إن \$ لا تكتب إلا في نهاية القواعد الخاصة بـ $\langle \text{exp} \rangle$ أي أننا لا يمكن أن نجدها في $\langle F \rangle$ أو في $\langle T \rangle$ أو في أي مكان آخر

لنكتب قاعدة $\langle T \rangle$ بشكل آخر حتى نتخلص من مشكلة بداية قاعدتين لها بنفس البداية:

$\langle T \rangle \rightarrow \langle F \rangle \langle T' \rangle$

$\langle T' \rangle \rightarrow \varepsilon$

$| * \langle T \rangle$

وكذلك نفعل مع $\langle \text{exp} \rangle$:

$\langle \text{exp} \rangle \rightarrow \langle T \rangle \langle T'' \rangle$

$\langle T'' \rangle \rightarrow \varepsilon$

$| + \langle \text{exp} \rangle$

والآن أصبح بإمكاننا كتابة المحلل القواعدي:

```
public class parser{
    private lexical lex;
    public parser(String f)
    {
        lex=new lexical(f);
    }
}
```

```
public boolean parser()
{
    return exp();
}

public boolean F(){
    Token t=lex.nextToken();
    switch(t.type){
        case Token.ID:
        case Token.INT_NUM:
            return true;
        case Token.LPAR:
            if(exp())
            {
                if (lex.nextToken().type == Token.RPAR)
                    return true;
                else
                    return false; // There is no closing parenthesis
            }
            else
                return false;
        }
    }
    return false;
}

private boolean T()
{
}
```

```
if(F())
{
    Token t=lex.nextToken();
    if(t.type!=Token.STAR)
    {
        lex.ungetToken();
        return true;
    }
    if(T())
        return true;
    else
        return false;
}
return false;
}

private boolean exp()
{
    if(T())
    {
        Token t=lex.nextToken();
        if(t.type!=Token.PLUS)
        {
            lex.ungetToken();
            return true;
        }
        if(exp())
            return true;
    }
}
```

```

else
    return false;
}
return false;
}
}

```

ملاحظة:

لاحظ أننا بحاجة لإجرائية `ungetToken` والتي تقوم بإعادة `token` كامل إلى الملف حتى نقوم بسحبه من جديد، وبما أن `token` قد يكون عادة مؤلف من عدة محارف فإن إعادته إلى الملف قد تكون معقدة قليلاً لذلك عادة ما تقوم إجرائية `nextToken` بإعادة الـ `token` الحالي بدون سحبه من الملف (`look ahead`) أي أن الاستدعاء المتتالي لـ `nextToken` يعطي نفس الـ `token` حتى نقوم باستدعاء إجرائية `advanced` التي تقوم بتحريك مؤشر الملف إلى `token` التالي وبالتالي يعطي استدعاء `nextToken` بعد إجرائية `advanced` الـ `token` التالي (`token` جديد)

:Predictive parsing table

– يمكننا من المثال السابق ملاحظة أنه من الأهمية بمكان أن نعرف ما هي القاعدة التالية التي يجب أن ننفذها انطلاقاً من القاعدة الحالية؟ (ما هو التابع الذي يجب أن نستدعيه انطلاقاً من التابع الحالي؟)

– فإذا كان لدينا عدة قواعد لـ `<exp>` مثلاً فأياً سنطبق حتى نولد السلسلة المطلوبة؟
 – الحل هو استخدام عدة طرق تعتمد على جدول يدعى جدول التحليل `predictive parsing table` حيث يكون لدينا علم مسبق من خلال هذا الجدول بالقاعدة التي يجب أن نطبقها الآن وذلك وفق `token` التالي
 بنية جدول التحليل:

– كما نعلم فإن لكل رمز غير نهائي تابع

– وبالمقابل فإن لكل تابع سطر خاص به في الجدول

– وكذلك فكل token لدينا عمود خاص به في الجدول

– عند تقاطع السطر والعمود نضع القاعدة التي يجب أن تطبق إذا كنا في التابع الخاص بهذا السطر وكان ال token التالي هو ال token الخاص بهذا العمود

مثال:

لنطبق ذلك على مثال $\langle \text{exp} \rangle$

	+	*	()	id	num	\$
$\langle \text{exp} \rangle$							
$\langle T \rangle$							
$\langle F \rangle$			$\langle F \rangle \rightarrow (\langle \text{exp} \rangle)$		$F \rightarrow \text{id}$	$F \rightarrow \text{num}$	

– لقد قمنا في الجدول السابق بملء بعض الحقول لفهم الفكرة فقط ولم نملء كامل حقول الجدول لأن الحقول الباقية ليست بديهية بل تحتاج لإجرائية خاصة لتقوم بتعبئتها وهي ما سنناقشه لاحقاً

– وإذا وجدنا خانات فارغة في الجدول بعد تطبيق الخوارزمية الخاصة بملء هذا الجدول عندها تمثل هذه الفراغات وجود مشكلة (حدوث خطأ) لأنها تقابل ورود رمز غير مقبول

ملاحظة:

تدعى هذه الطريقة بـ predictive parsing

:LL(1)

– تدعى القواعد التي لا تحتوي في جدول التحليل predictive parsing table على أكثر من قيمة واحدة في كل خانة بقواعد LL(1) أو left-to-right parse أو leftmost-derivation أو 1-symbol look ahead

– نقوم هنا بالسير على سلسلة ال token من اليسار إلى اليمين

– أي أن الشجرة تبنى من الرموز غير النهائية وذلك من اليسار إلى اليمين

– هنا نقوم بمعرفة القاعدة التي يجب أن نطبقها (التابع التالي الذي سيتم استدعاءه) من خلال معرفتنا لـ token واحد أمامنا فقط

– كما يمكننا أن نقوم بتعميم $LL(1)$ إلى $LL(k)$ حيث k أي رقم صحيح أكبر تماماً من الصفر، حيث نتعرف على القاعدة التي يجب أن نطبقها بعد معرفتنا لـ k token التالية، ولكن ذلك لا يطبق بشكل واسع لأن أعمدة جدول التحليل ستتحول عندها لتحتوي على كل التراكيب الممكنة لـ k من الـ tokens وهذا يجعل الجدول ضخماً جداً

فمثلاً نحتاج لمعرفة أكثر من token واحد حتى نستطيع تحديد القاعدة التالية لـ $\langle s \rangle$ في حال وجود القواعد:

$\langle s \rangle \rightarrow ab\langle x \rangle$
 $\quad \quad \quad | \quad ab\langle y \rangle$

ملاحظة:

لا يستخدم عادة محلل القواعد من النوع LL وذلك لأنه ليس قوي بدرجة كافية، بل يستخدم بدل منه محلل القواعد من النوع LR لأنه يعتمد على stack يتم حشر الـ tokens فيه ثم وعندما نلاحظ اكتمال قاعدة ما نقوم باستكمالها بالرمز الموجود على يسارها (عند اكتمال $\langle exp \rangle$) نقوم باستبدالها كلها بـ $\langle F \rangle$ ، وسبب قوة LR هو وجود بعض القواعد التي لا نستطيع كتابتها بشكل LL أبداً ولكننا نستطيع استخدامها في حالة LR

الطريقة العامة لـ $LL(1)$:

❖ نقوم بإنشاء تابع خاص بكل رمز غير نهائي

❖ يقوم هذا التابع بالحصول على token التالية (token واحدة فقط)

❖ يعرف من خلال جدول التحليل التابع الذي يجب أن يستدعيه وذلك بالاستعانة بالـ token الذي حصل عليه

ملاحظة:

لاحظ سهولة تعميم هذه الطريقة للحصول على الطريقة العامة لـ $LL(k)$

خوارزمية إنشاء جدول تحليل LL(1):

تعتمد هذه الخوارزمية على 3 مفاهيم (مجموعات) تخص الرموز غير النهائية بشكل عام:

١. المجموعة **first**:

هي مجموعة كل الرموز النهائية التي يمكن أن تبدأ بها سلاسل المفردات المولدة من هذا الرمز غير النهائي

ملاحظة: على الرغم من أن مجموعة first تخص عادة الرموز غير النهائية إلا أنه يمكننا أن نقول أن المجموعة first للرمز النهائي مكونة من عنصر واحد هو نفس هذا العنصر النهائي أي:

$$\text{first}(a) = \{a\}$$

مثال:

لنفترض القواعد التالية:

$$\begin{aligned} \langle s \rangle &\rightarrow a \\ &\quad | \quad b\langle x \rangle \\ &\quad | \quad \langle x \rangle \\ \langle x \rangle &\rightarrow c \\ &\quad | \quad \varepsilon \end{aligned}$$

عندها يكون:

$$\begin{aligned} \text{first}(x) &= \{c\} \\ \text{first}(s) &= \{a, b, c\} \end{aligned}$$

٢. المجموعة **follow**:

هي مجموعة كل الرموز النهائية token التي يمكن أن تأتي مباشرة بعد الرمز غير النهائي الحالي في اشتقاق derivation ما

الاشتقاق derivation:

هو أي سلسلة من الرموز النهائية وغير النهائية نحصل عليها انطلاقاً من s (تذكر أن s هو رمز البداية، أي أنه القاعدة الأولى التي نطبقها (جذر شجرة التوليد))

ملاحظة:

إن هذه المجموعة أيضاً تخص الرموز غير النهائية، على الرغم من أننا نستطيع أن نوجدها من أجل الرموز النهائية tokens ولكن ذلك لن يفيدنا بشيء

مثال:

$$\langle s \rangle \rightarrow \dots \rightarrow \dots \rightarrow \underline{a} \underline{b} \langle x \rangle \underline{c} \underline{b} \langle y \rangle \underline{z}$$

عندها يمكننا أن نستنتج أن:

$$c \in \text{follow}(x), z \in \text{follow}(y)$$
مثال:

$$\langle s \rangle \rightarrow \dots \rightarrow \dots \rightarrow \underline{a} \underline{b} \langle x \rangle \langle y \rangle \underline{z}$$

إذا كان $\langle y \rangle$ يمكن أن يتحول إلى null عندها يمكننا أن نستنتج أن $z \in \text{follow}(x)$

بينما إذا لم تكن $\langle y \rangle$ قابلة للتحويل إلى null (nullable) عندها لا يمكننا أن نستنتج ذلك

ملاحظة:

إذا كان لدينا:

$$\langle s \rangle \rightarrow \dots \rightarrow \dots \rightarrow \langle x \rangle \langle y \rangle \langle z \rangle$$

وكان $\langle y \rangle$ قابلة لأن تصبح null عندها يمكننا ببساطة أن نضيف $\text{first}(z)$ إلى $\text{follow}(x)$

٣. المجموعة **nullable**:

وهي مجموعة واحدة تضم جميع الرموز غير النهائية التي من الممكن أن نشق الخالية (ϵ) منها

ملاحظة: إن هذه المجموعة خاصة بالرموز غير النهائية ولكن يمكننا القول تجاوزاً أن الرموز النهائية ليست nullable لأننا لا نستطيع أن نشق منها أي شيء بما فيها الخالية (ϵ)

مثال:

إذا كان لدينا:

$$\begin{aligned} \langle z \rangle &\rightarrow \langle x \rangle \langle y \rangle \\ &| a \langle x \rangle \langle f \rangle \\ \langle x \rangle &\rightarrow \langle y \rangle \\ &| b \\ \langle y \rangle &\rightarrow \epsilon \\ &| c \\ \langle f \rangle &\rightarrow ab \\ &| c \end{aligned}$$

عندها يمكننا أن نستنتج أن:

nullable(y) = true
nullable(x) = true
nullable(z) = true
nullable(f) = false

That 's all folks



lectures_team@hotmail.com

طريقة LL(1)

الاسم: ١٦/١٠/٢٠٠٦

خوارزمية حساب المجموعات **first** و **follow** و **nullable**:

- سنكتب الآن خوارزمية واحدة تقوم بإيجاد هذه المجموعات الثلاثة

```

for each terminal symbol Z      FIRST[Z] ← {Z}
for each non-terminal symbol Z  nullable[Z] ← false; follow[Z]=φ ; first[Z]=φ

while not stable //FIRST, FOLLOW, or nullable is changed in this iteration.
  for each production X → Y1Y2 ... Yk
    if Y1... Yk are all nullable (or if k = 0)
      then nullable[X] ← true
    for each i from 1 to k, each j from i + 1 to k
      if Y1 ... Yi-1 are all nullable (or if i = 1)
        then FIRST[X] ← FIRST[X] ∪ FIRST[Yi]
      if Yi+1 ... Yk are all nullable (or if i = k)
        then FOLLOW[Yi] ← FOLLOW[Yi] ∪ FOLLOW[X]
      if Yi+1 ... Yj-1 are all nullable (or if i + 1 = j)
        then FOLLOW[Yi] ← FOLLOW[Yi] ∪ FIRST[Yj]

```

ملاحظات:

◀ نلاحظ أن الخوارزمية السابقة تقوم بإيجاد المجموعات الثلاثة معاً فهي لا تتوقف إلا إذا لم تتغير أي من هذه المجموعات الثلاثة طوال تكرار iteration كامل

◀ كان يمكننا أن نكتب إجرائية خاصة بإيجاد nullable لأنها لا تعتمد على أي مجموعة أخرى ثم نكتب إجرائية لحساب first التي لا تعتمد إلا على nullable ثم نكتب إجرائية لحساب follow التي تعتمد على nullable و follow

◀ نلاحظ أن الخوارزمية بسيطة لأنها تعتمد على ثلاثة حالات:

- بداية أحد الاشتقاقات derivation هي nullable
- نهاية أحد الاشتقاقات derivation هي nullable
- منتصف أحد الاشتقاقات derivation هو nullable

مثال:

لنفترض القواعد التالية:

$$\begin{aligned}
 Z &\rightarrow d & Z &\rightarrow X Y Z \\
 Y &\rightarrow \varepsilon & Y &\rightarrow c \\
 X &\rightarrow Y & X &\rightarrow a
 \end{aligned}$$

وعند تنفيذ الخوارزمية السابقة نجد أن جدول التحليل تتم تعبئته وفق الترتيب التالي (يمثل الرقم العلوي

ترتيب ملء حقول الجدول):

	nullable	first	follow
X	yes ⁴	a ⁶ c ⁵	a ⁹ d ⁹ c ⁹
Y	yes ²	c ³	a ¹⁰ d ¹⁰ c ¹⁰
Z	no	a ⁸ d ¹ c ⁷	

توسعة:

- لا بد لنا من القيام بتوسعة بعض المفاهيم من أجل إنشاء جدول التحليل

- سنقوم الآن بتوسعة مفهوم nullable و first ليشمل السلاسل، أي طريقة تحديد هل السلسلة هي

nullable أم لا وكذلك إيجاد first الخاص بسلسلة

توسعة nullable:

يكون nullable(γ) = true إذا كانت γ مكونة من رموز غير نهائية كلها nullable

توسعة first:

$$\begin{aligned}
 \text{first}(X\gamma) &= \text{first}(X) \text{ if } X \text{ is not nullable} \\
 &= \text{first}(X) \cup \text{first}(\gamma) \text{ if } X \text{ is nullable}
 \end{aligned}$$

مثال: من قواعد المثال السابق يمكننا كتابة:

$$\begin{aligned}
 \text{first}(XYZ) &= \text{first}(X) \cup \text{first}(YZ) \\
 &= \text{first}(X) \cup \text{first}(Y) \cup \text{first}(Z)
 \end{aligned}$$

وذلك لأن كل من X و Y هو nullable

خوارزمية إيجاد جدول التحويل:

١. ننشئ جدولاً فارغاً أسطره الرموز غير النهائية وأعمدته الرموز النهائية

٢. من أجل كل قاعدة $\gamma \rightarrow X$ نضيف هذه القاعدة في السطر X وفي كل عمود T ينتمي إلى $\text{first}(\gamma)$ وإذا كانت γ هي nullable نضيف هذه القاعدة أيضاً في السطر X وكل عمود T ينتمي إلى $\text{follow}(X)$

ملاحظة: لاحظ أننا احتجنا هنا لتوسعة first و nullable لتشمل سلسلة

ففي المثال السابق يكون الجدول بالشكل:

	a	c	d
X	$X \rightarrow Y$ $X \rightarrow a$	$X \rightarrow Y$	$X \rightarrow Y$
Y	$Y \rightarrow \epsilon$	$Y \rightarrow c$ $Y \rightarrow \epsilon$	$Y \rightarrow \epsilon$
Z	$Z \rightarrow XYZ$	$Z \rightarrow XYZ$	$Z \rightarrow d$ $Z \rightarrow XYZ$

ملاحظات:

- الخانات الفارغة في الجدول هي أخطاء، لأنها تعني أنه لا يوجد أي قاعدة نقوم بتطبيقها
- إن وجود قاعدتين في خانة واحدة يعني أن القواعد غامضة (لأننا لن نعرف أي قاعدة يجب أن ننفذ من هاتين القاعدتين) أي أنها ليست LL(1) ولا يمكن إنشاء جدول تحليل LL(1) لها
- في حالة LL(2) يتحول اسم كل عمود إلى رمزين نهائيين وتصبح الأعمدة هي كل التشكيلات الممكنة لرمزين نهائيين

بعض المشاكل الممكن حلها لتحويل قواعد ليست LL(1) إلى قواعد LL(1):

❖ العامل المشترك الأيسر بين قاعدتين:

$S \rightarrow \underline{\text{if } E \text{ then } S} \text{ else } S$
| $\underline{\text{if } E \text{ then } S}$

يمكننا حل هذه المشكلة بفرض قاعدة جيدة تحتوي الجزء اليساري المشترك بين القاعدتين وقاعدة خاصة بالجزئين المتبقين:

$S \rightarrow \underline{\text{if } E \text{ then } S} S_2$
 $S_2 \rightarrow \underline{\text{else } S}$
| ϵ

❖ العودية اليسارية:

حالة عامة:

$$\left. \begin{array}{l} X \rightarrow X \gamma_1 \\ X \rightarrow X \gamma_2 \\ X \rightarrow \alpha_1 \\ X \rightarrow \alpha_2 \end{array} \right\} \Rightarrow \begin{array}{l} X \rightarrow \alpha_1 X' \\ X \rightarrow \alpha_2 X' \\ X' \rightarrow \gamma_1 X' \\ X' \rightarrow \gamma_2 X' \\ X' \rightarrow \varepsilon \end{array}$$

مثال:

$$\begin{array}{l} E \rightarrow E \pm T \\ \quad | \quad T \end{array}$$

تصبح بعد التخلص من العودية:

$$\begin{array}{l} E \rightarrow T E_2 \\ E_2 \rightarrow \pm T E_2 \\ \quad | \quad \varepsilon \end{array}$$

معالجة الأخطاء:

- من جدول التحليل يمكننا أن نستنتج ببساطة وجود switch على token في كل تابع
- في default الخاصة بهذه ال switch نقوم بمعالجة الخطأ (حالة فراغ في جدول التحليل)

طرق معالجة الأخطاء:

- ◀ إيقاف التحليل فوراً بعد طباعة رسالة خطأ
- ◀ طباعة رسالة خطأ ومحاولة التصحيح ومتابعة التحليل

ملاحظة:

دائماً يجب إيقاف توليد ال code عند حدوث خطأ في compiler

تصحيح الخطأ:

يمكننا تصحيح الخطأ عن طريق الحذف أو عن طريق الإضافة

مثال:

بالرجوع إلى قواعد التعبير expression يمكننا تذكر وجود القاعدة:

$$f \rightarrow \underline{id} \mid \underline{num} \mid (E)$$

فعند كتابة تابع f يمكننا أن نكتب:

void f(){

```

switch token.type{
    case ID: case num: return true;
    case LPAR: /*some necessary code*/ return true;
    default: /*print error message*/
}
return true;
}

```

فهنا عندما لا نجد أي من id أو num أو lpar نقوم بإظهار رسالة خطأ كما أننا نكون قد افترضنا وجود id ونكمل بعد ذلك بدون أن يقوم التابع الأب الذي يستدعي هذا التابع بأي عمل زائد لأنه لن يشعر أصلاً بوجود خطأ بسبب إعادة التابع f للقيمة true إذا أردنا التصحيح بالحذف فإننا نضيف حلقة تقوم بقراءة tokens حتى نصل إلى token ينتمي إلى follow(f) عندها نتوقف

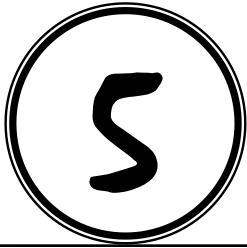
مقارنة بين تصحيح الخطأ بالإضافة أو الحذف:

التصحيح بالإضافة قد لا ينتهي لأننا قد نحصل على خطأ نتيجة التصحيح فنضطر لإضافة جديدة لحل هذا الخطأ وهكذا ... ، مما يسبب ظهور الكثير من الأخطاء الوهمية التي لا وجود لها عملياً والتي تزول بإزالة الخطأ الأصلي، هذا بالإضافة إلى أن التصحيح بالإضافة يتطلب إعادة token الذي تم قراءته إلى الملف

That 's all folks



lectures_team@hotmail.com



الترجمات



طريقة LR

الاثنين ٣٠/١٠/٢٠٠٦

مقارنة LR مع LL(1):

- في LL(1) كنا نطبق القواعد من اليسار إلى اليمين (left most derivation)

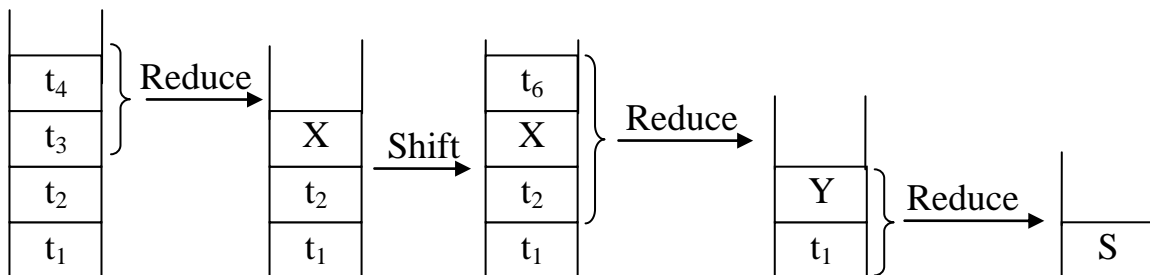
- بينما هنا نطبق القواعد من اليمين إلى اليسار (right most derivation)

- أي أننا نقوم بتجميع الـ tokens في stack وعندما نحصل على مجموعة من tokens مطابقة لطرف يميني لإحدى القواعد الموجودة لدينا فإننا نقوم باستبدال هذه المجموعة بالطرف اليساري للقاعدة المقابلة

مثال:

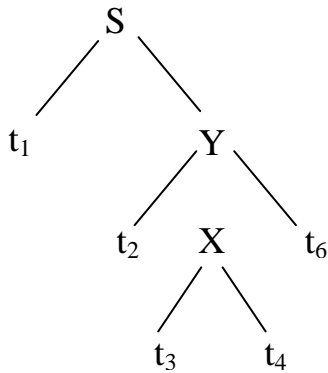
لنفرض أن لدينا القواعد التالية:

$$\begin{aligned} X &\rightarrow t_3 t_4 \\ Y &\rightarrow t_2 X t_6 \\ S &\rightarrow t_1 Y \end{aligned}$$

وأن سلسلة الدخل هي: $t_1 t_2 t_3 t_4 t_6$ ملاحظة: القرارات السابقة (reduce, shift) يتم اتخاذها وفق جدول سنقوم بدراسته لاحقاً

- نقول عن سلسلة مفردات الدخل أنها صحيحة قواعدياً إذا احتوى المكس عند نهاية سلسلة الدخل على رمز البداية (S)

- لاحظ أننا إذا بنينا شجرة التحليل هنا فإننا نقوم ببنائها من الأسفل للأعلى



أي يتم بناء الشجرة المجاورة من الأسفل للأعلى:
إذا فإن محلل LR يعتمد على جدول تحليل LR

جدول التحليل LR:

لنفرض الآن أننا حصلنا على جدول تحليل LR بطريقة ما (هناك خوارزمية لإنشاء هذا الجدول سندرسها لاحقاً) ولنقوم الآن بفهم هذا الجدول وآلية التعامل معه

مثال:

لنفرض أن لدينا القواعد التالية:

0. $S' \rightarrow S\$$
1. $S \rightarrow S _ S$
2. $S \rightarrow \text{id} _ = E$
3. $S \rightarrow \text{print} (_ L)$
4. $E \rightarrow \text{id}$
5. $E \rightarrow \text{num}$
6. $E \rightarrow E _ + E$
7. $E \rightarrow (_ S _ E)$
8. $L \rightarrow E$
9. $L \rightarrow L _ E$

ولنفرض أن سلسلة الدخل هي:

a := 7;

b := c +(d := s+6 , d)

ملاحظة: إن العبارة (d := s+6 , d) يقوم الحاسب بتنفيذها من اليسار لليمين ثم يعيد ناتج آخر عملية موجودة فيها (العملية الموجودة بعد آخر فاصلة) وهذا الشيء موجود حتى في لغات البرمجة مثل ++c التي تسمح بها كما عندما نكتب في حلقة الـ for:

for(int i=1; j=2; i<10; i++, j--)

ولكننا هنا لا نستفيد من القيمة المعادة من العملية الموجودة بعد الفاصلة الأخيرة
هنا سيكون جدول التحليل على الشكل:

شرح رموز الجدول:

الحالة



رموز نهائية

رموز غير نهائية

	id	num	print	:	,	+	:=	()	\$	S	E	L
1	s4		s7								g2		
2				s3						a			
3	s4		s7								g5		
4						s6							
5				r1	r1					r1			
6	s20	s10					s8				g11		
7							s9						
8	s4		s7								g12		
9	s20	s10					s8				g15	g14	
10				r5	r5	r5		r5	r5				
11				r2	r2	s16			r2				
12				s3	s18								
13				r3	r3				r3				
14					s19			s13					
15					r8			r8					
16	s20	s10					s8				g17		
17				r6	r6	s16		r6	r6				
18	s20	s10					s8				g21		
19	s20	s10					s8				g23		
20				r4	r4	r4		r4	r4				
21								s22					
22				r7	r7	r7		r7	r7				
23					r9	s16		r9					

يحتوي فقط على
(goto) g

❖ s4: قم بعملية shift (push في stack والانتقال إلى token التالي) واذهب للحالة رقم 4

❖ r1: قم بعملية reduce (استبدال عدة عناصر على قمة المكس بالطرف اليساري للقاعدة

الخاصة بهم) حسب القاعدة رقم 1 وستخبرنا goto (الموجودة في الأعمدة الخاصة بالرموز غير النهائية) بعد ذلك برقم الحالة التي يجب أن نذهب لها

❖ goto: g2 اذهب للحالة رقم 2

❖ a: قبول سلسلة الدخل

إذا لا actions المتاحة هي:

Shift(s), Reduce(r), Goto(g), Accept(a)

ملاحظة: لا نجد accept إلا عند العمود الخاص بـ \$ لأنها تدل على قبول سلسلة الدخل والذي لا يتم

إلا عند انتهاء ملف الدخل (أي ورود \$)

والآن لننفذ الدخل السابق بالاستعانة بهذا الجدول:

ملاحظات:

-في كل خطوة ننظر إلى الخانة المناسبة من الجدول وننفذ ما بداخلها
 -سنقوم بعملية push لرقم الحالة مع ال token في نفس المكس بدل إنشاء مكس مستقل لأرقام الحالات

-قمة المكس تحوي دائماً الحالة الحالية

الخطوة 1:

1

a:= 7;...

سيكون في tokens stack الرقم 1 لأننا نبدأ من الحالة رقم 1 ونحن عند الرمز a(id) من الدخل عندها ننظر إلى السطر رقم 1 والعمود الخاص ب id فنجد s4 أي أننا نقوم ب shift ونذهب للحالة 4

الخطوة 2:

4
id
1

a:= 7;...

الحالة الحالية هي الحالة رقم 4 ولدينا الرمز := في الدخل فيكون الرمز المقابل من الجدول هو s6 أي أننا نقوم ب shift ونذهب للحالة 6

الخطوة 3:

6
:=
4
id
1

a:= 7;...

الحالة الحالية هي الحالة رقم 6 ولدينا الرمز 7(num) في الدخل فيكون الرمز المقابل من الجدول هو s10 أي أننا نقوم ب shift ونذهب للحالة 10

الخطوة 4:

10
num
6
:=
4
id
1

a:= 7;...

الحالة الحالية هي الحالة رقم 10 ولدينا الرمز ؛ في الدخل فيكون الرمز المقابل من الجدول هو r5 (E → num) أي أننا نقوم ب reduce وفق القاعدة رقم 5

آلية القيام ب **reduce**: نقوم بعملية pop من المكس بعدد يساوي ضعف عدد العناصر الموجودة في الجزء اليميني من القاعدة رقم 5 لأننا في كل مرة سنزيل إما أحد هذه العناصر من المكس أو رقم حالة ونقوم

بعدها بإضافة الرمز الموجود على اليسار في القاعدة، ولمعرفة رقم الحالة الجديد فإننا نقوم بالنظر إلى الرمز الجديد الذي أضفناه إلى قمة المكس (الرمز الموجود في الجزء اليساري من القاعدة) بالإضافة

إلى رقم الحالة الموجودة تحته تماماً و ننتقل للجدول حيث سنجد حتماً تعليمة goto لأننا سننظر إلى الجزء الخاص بالرموز غير النهائية فيه

أي أننا هنا نقوم بإزالة 10 و num من قمة المكس ونضع بدل منها E ثم نذهب إلى الخانة الخاصة بالسطر رقم 6 والعمود E فنجد g11 من هنا نستنتج أن الحالة التالية هي 11 فنقوم بإضافة 11 إلى قمة المكس

الخطوة 5:

11
E
6
:=
4
id
1

↓
a:= 7;...

الحالة الحالية هي الحالة رقم 11 ولدينا الرمز ؛ في الدخل (لم يتغير رمز الدخل لأننا لم نقوم بـ shift عليه) فيكون الرمز المقابل من الجدول هو $r2 (S \rightarrow id := E)$ أي أننا نقوم بـ reduce وفق القاعدة رقم 2 فنقوم بحذف 6 عناصر من قمة المكس ونضع بدل منهم S ثم ننظر لـ S ورقم الحالة الموجود أسفل منها والذي هو 1 فننتقل إلى الجدول لنجد g2 أي أننا نضيف على قمة المكس 2 (رقم الحالة الجديدة)

الخطوة 6:

2
S
1

↓
a:= 7;...

الحالة الحالية هي الحالة رقم 2 ولدينا الرمز ؛ في الدخل (لم يتغير رمز الدخل لأننا لم نقوم بـ shift عليه) فيكون الرمز المقابل من الجدول هو s3 أي أننا نقوم بـ shift لكل من ؛ و 3 فيصبح المكس كما في الشكل المجاور

3
;
2
S
1

↓
a:= 7;...

ونتابع هكذا...

ملاحظة: إن الخانات الفارغة في الجدول تمثل أخطاء

لنناقش الآن خوارزمية التحليل (خوارزمية التنفيذ بالاستعانة بالجدول السابق)

ملاحظة: قبل تنفيذ هذه الخوارزمية يجب أن نضع في المكس الرقم 1 خوارزمية التحليل:

(١) ليكن t هو الرمز الحالي من الدخل و q هي رقم الحالة على قمة المكس

(٢) إذا كان محتوى السطر q والعمود t هو:

push(t) :S(n) .a

push(n)

t = nextToken()

goto 2

pop :R(k) .b بعدد رموز الطرف الأيمن للقاعدة رقم k مضروبة بـ 2

ليكن x الرمز الأيسر للقاعدة رقم k

ننظر للسطر q₂ والعمود x من الجدول فنجد g(y)

push(x)

push(y)

a .c: انتهى التحليل بنجاح

d . فارغ: يوجد خطأ

توليد الجدول من القواعد:

– إن توليد جدول LR(0) فإننا لا ننظر للأمام أبداً لذلك فإن الطريقة LR(0) تعد ضعيفة ولكنها

أساس لـ LR(1) (بينما في LR(1) فقد ننظر للأمام فنستهلك الرمز التالي ونسير فوقه أو نقوم

برؤيته فقط بدون سحبه من الدخل)

– يجب أن نقوم ببناء أوتومات خاص ينتج عنه جدول التحليل (يشبه الأوتومات المنتظم)

بناء الأوتومات:

لنقوم بفهم آلية بناء هذا الأوتومات من خلال المثال التالي:

0. $S' \rightarrow S\$$
1. $S \rightarrow (L)$
2. $S \rightarrow \underline{x}$
3. $L \rightarrow S$
4. $L \rightarrow L _ S$

لبناء الأوتومات نقوم بما يلي:

١. نقوم بوضع حالة خاصة بالقاعدة رقم 1 ($S \rightarrow S \$$)

٢. نضع نقطة قبل S الموجودة على الطرف اليميني ($S \rightarrow .S \$$) مع ملاحظة أن النقطة تدل على

المكان الذي وصلنا إليه ضمن القاعدة

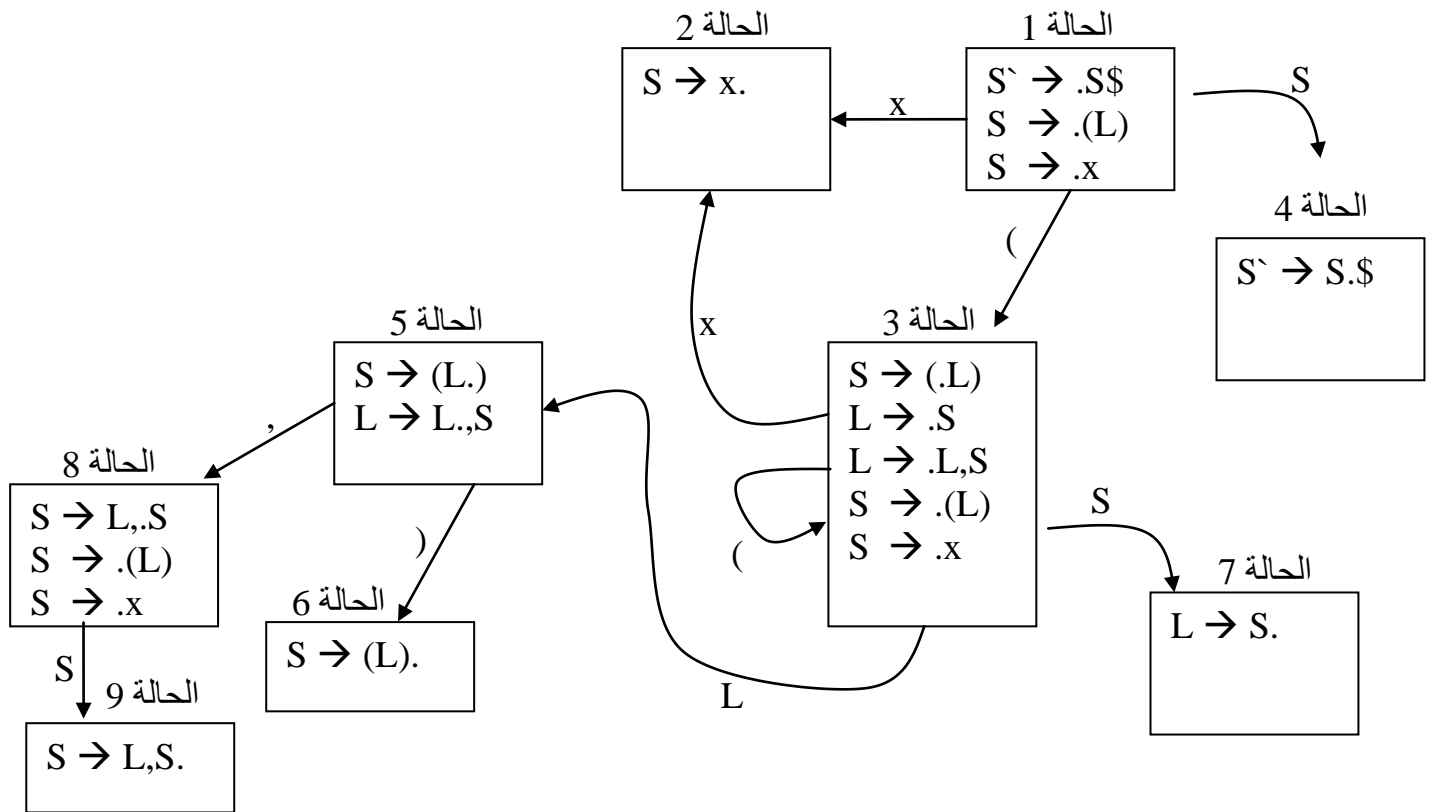
٣. نقوم من أجل حالة ما بتطبيق closure (إغلاق) عليها وذلك بأخذ القواعد التي تبدأ بالرموز غير النهائية التي قبلها نقطة ونضعها ضمن هذه الحالة وهكذا من أجل القواعد الجديدة التي أضيفت على الحالة حتى تستقر الحالة (ففي مثالنا السابق نضع قواعد S كلها ضمن الحالة الأولى أي $(L) \rightarrow S$ و $x \rightarrow S$ ولو كان x رمز غير نهائي لقمنا بوضع قواعده أيضاً ضمن الحالة الأولى)

٤. نقوم من أجل حالة ما بأخذ كل القواعد التي فيها نقطة وبعدها رمز نهائي أو غير نهائي إنشاء حالة جديدة لكل قاعدة منهم (ووضع هذا الرمز على الوصلة بين الحالة القديمة والحالة الجديدة) ووضع هذه القاعدة فيها مع تحريك النقطة لبعد هذا الرمز (مثلاً إنشاء حالة جديدة مرتبطة بسهم عليه " ووضع $(L) \rightarrow S$ فيها) ثم تطبيق الخطوة 3 عليها

٥. وهكذا حتى يستقر الأوتومات على الحالات النهائية له

ملاحظة: النقطة ليست رمز نهائي وليست token بل هي فقط دلالة على مكان وصولنا في القاعدة

ملاحظة: عادة لا نكمل من الحالة التي تحتوي على القاعدة $S \rightarrow S.$ (الحالة رقم 4 في الشكل التالي) لأننا نعتبر أن الحالة التالية لها $(S \rightarrow S.)$ موجودة بشكل دائم



That's all folks

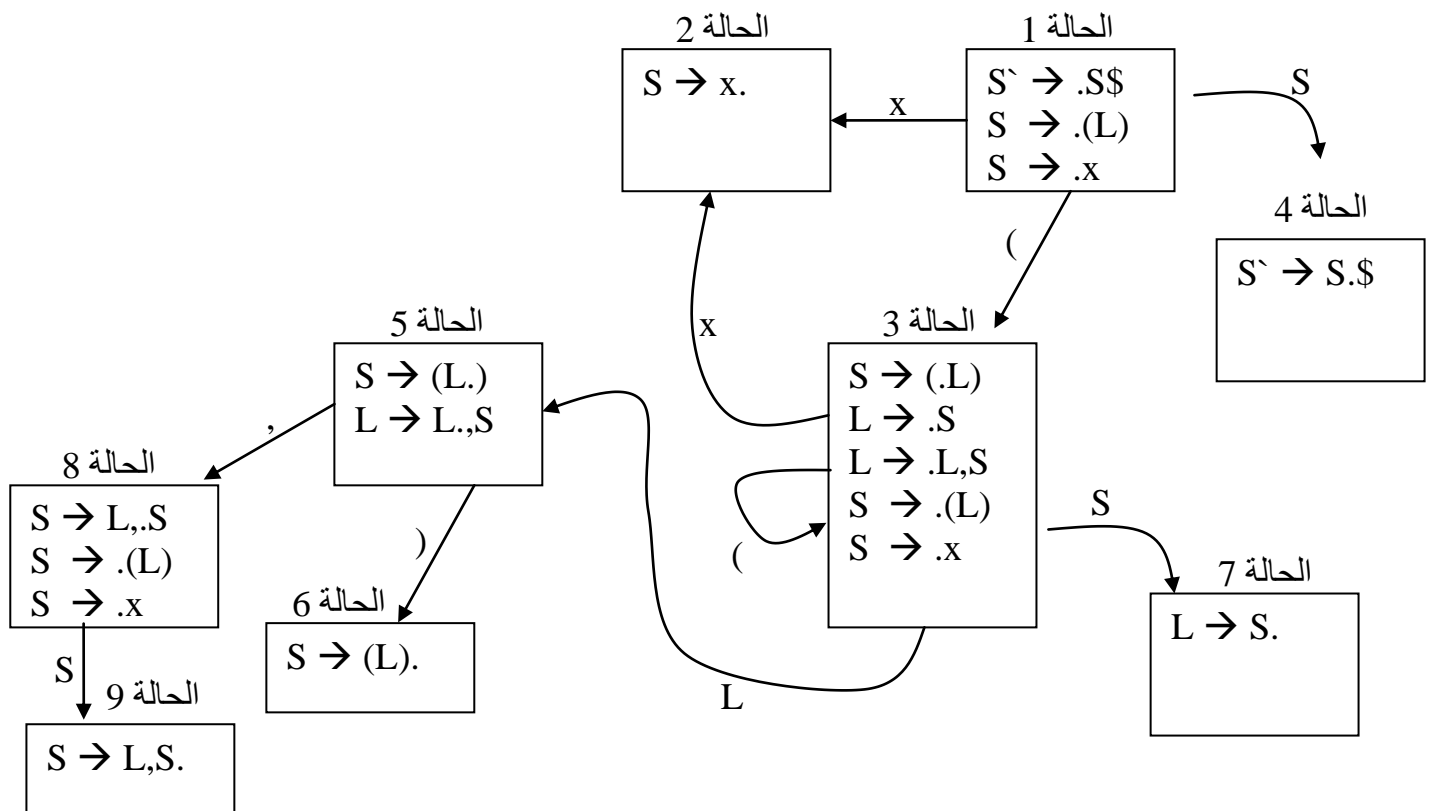


lectures_team@hotmail.com

تتمة الطريقة LR

الاثنين ١٣/١٠/٢٠٠٦

كنا في المحاضرة السابقة قد توصلنا إلى الأوتومات الذي سنشتق منه جدول التحليل الخاص بالطريقة LR وتركنا طريقة الاشتقاق لهذه المحاضرة طريقة اشتقاق جدول التحليل من الأوتومات:



كنا في المحاضرة السابقة قد توصلنا للأوتومات السابق من أجل القواعد:

0. $S' \rightarrow SS$
1. $S \rightarrow (L)$
2. $S \rightarrow x$
3. $L \rightarrow S$
4. $L \rightarrow L, S$

- عدد أسطر الجدول هو عدد حالات الأوتومات
- نقوم بتعبئة الجدول حسب الأسهم الخاصة بالأوتومات:

• إذا كان لدينا: $3 \xrightarrow{\text{رمز غير نهائي } L} 5$ عندها نضع في السطر 3 والعمود L نضع g5

• إذا كان لدينا: $1 \xrightarrow{\text{رمز نهائي } 'x'} 2$ عندها نضع في السطر 1 والعمود x نضع s2

• الحالة التي تحتوي على قاعدة فيها \$. نضع في سطرها والعمود \$ القيمة a لأنها تعني قبول السلسلة

- من أجل reduce نقوم بأخذ كل حالة تحتوي على $s \rightarrow x$ (نقطة في نهاية قاعدة) نضع في سطرها وجميع الأعمدة المقابلة لرمز نهائي (حتى \$) reduce بالرقم المقابل للقاعدة $s \rightarrow x$
- ملاحظات:** لاحظ أننا لا نضع g إلا من أجل السهم ذو الرمز غير النهائي وهذا ما يفسر وجود g في الأعمدة المقابلة للرموز غير النهائية في الجدول فقط من أجل المثال السابق نحصل على الجدول التالي:

	()	x	.	\$	S	L
1	s3		s2			g4	
2	r2	r2	r2	r2	r2		
3	s3		s2			g7	g5
4					a		
5			s6		s8		
6	r1	r1	r1	r1	r1		
7	r3	r3	r3	r3	r3		
8	s3		s2			g9	
9	r4	r4	r4	r4	r4		

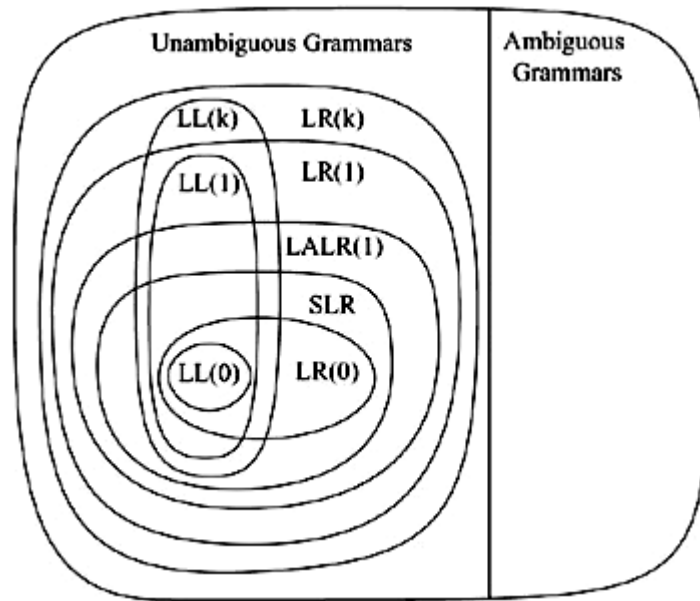
ملاحظة: إذا كان لدينا في كل خانة من خانات هذا الجدول قيمة واحدة تكون القواعد التي انطلقنا منها

هي قواعد LR(0)، وإلا إذا حصل تضارب (وجود قيمتين في نفس الخانة) فإننا نكون مضطرين

لتطبيق خوارزمية أفضل قليلاً للتخلص من التضارب هي SLR

ملاحظة: إن SLR هي أفضل من LR(0) ولكنها تبقى أسوأ من LR(1) والتي غالباً ما تكفي في أية

لغة برمجة

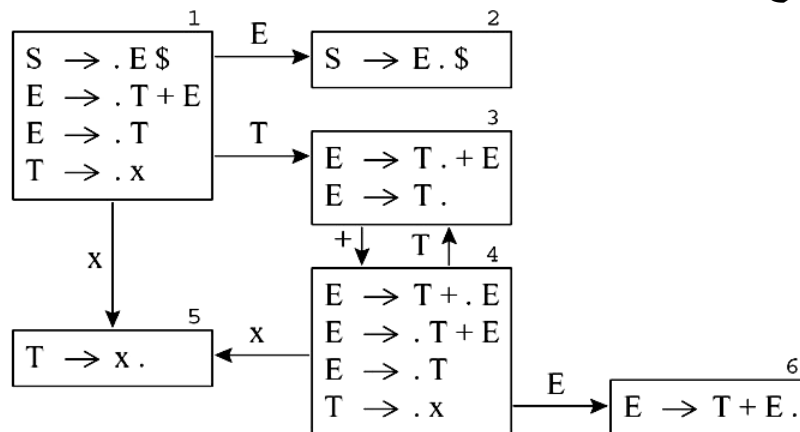


طريقة SLR:

إذا حدث تعارض في إحدى خانات جدول التحليل عندها نلجأ لطريقة أقوى هي SLR ولنأخذ مثال على حالة تعارض تستطيع SLR أن تقوم بحلها

0. $S \rightarrow E \$$
1. $E \rightarrow T \pm E$
2. $E \rightarrow T$
3. $T \rightarrow \underline{x}$

يمكننا ببساطة أن نستنتج الأوتومات الخاص بهذه القواعد وجدول التحليل:



	x	+	\$	E	T
1	s5			g2	g3
2			a		
3	r2	s4,r2	r2		
4	s5			g6	g3
5	r3	r3	r3		
6	r1	r1	r1		

- لاحظ وجود تعارض عند الحالة 3 والعمود +: فبما أن لدينا سهم من الحالة 3 إلى الحالة 4 فلا بد من وجود s4 في السطر 3 والعمود +، وبما أن لدينا $E \rightarrow T$ ضمن القاعدة 3 فلا بد من وضع r2 (القاعدة $E \rightarrow T$ هي القاعدة رقم 2) في السطر 3 والعمود +

- لحل هذه المشكلة يمكننا تغليب s على r فنحذف r في هذه الحالة من السطر 3 والعمود + ونترك s4 (في غالبية الحالات يكون هذا الحل صحيح كما هو في مثالنا الحالي حيث يتعرف الـ compiler على اللغة بشكل سليم) ولكن الحل الأفضل هو تغيير القواعد

- لحل هذه المشكلة باستخدام SLR نقوم بتعديل طفيف على طريقة إضافة reduce إلى الجدول حيث نأخذ الحالة التي تحتوي على $S \rightarrow T$ ونضع r مع رقم القاعدة $S \rightarrow T$ في السطر الخاص بهذه الحالة والأعمدة التي تنتمي إلى follow(S) فقط، ففي مثالنا هذا $\text{follow}(E)$ هو \$ فقط ← نضع r2 في العمود \$ فقط من السطر 3

خوارزمية LR(1):

تشبه إلى حد بعيد LR(0) لذلك لن نشرحها هنا

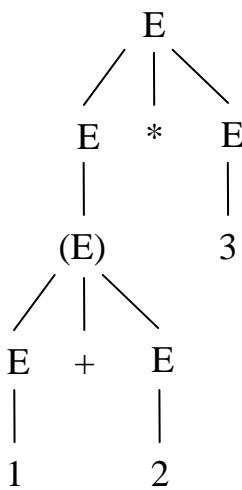
شجرة التحليل المجردة Abstract Syntax

- شجرة التحليل التي نريد الحصول عليها تشبه شجرة التحليل للقواعد خارج السياق
- نريد الحصول على شجرة تحليل تجنبنا مشاكل الأقواس وأفضليات العمليات
- قديماً كنا لا نولد شجرة تحليل لأن الحواسيب بطيئة والذاكر قليلة المساحة مما كان يضطرنا لإنشاء compiler بمسحة واحدة على الـ code ويجعلنا لا نولد أي شكل من أشكال التمثيل الوسيطي (مثل شجرة التحليل)، مما كان يسبب حشر كامل الـ code في ملف واحد بالإضافة لتراكب كل المراحل التي تلي مرحلة التحليل مع هذه المرحلة حيث كنا نضطر لكتابة التحقق من الأنماط هنا وكذلك توليد الـ code النهائي (قد يصل حجم الـ action الواحد إلى عدة صفحات!!)

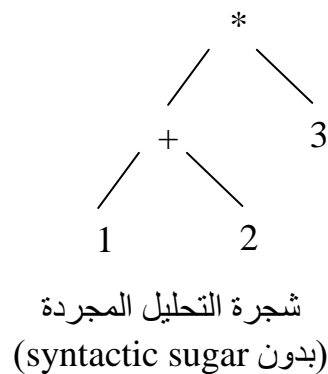
-بينما حالياً أصبح لدينا حواسيب سريعة وذواكر كبيرة مما سمح لنا بإنشاء compiler بعدة مسحات على الـ code وسمح لنا بتقسيم الـ compiler إلى modules حيث نقوم بإنشاء شجرة التحليل بعد مرحلة التحليل والتي تمرر لباقي المراحل (قد نقوم ببعض التحسينات قبل أو بعد بناء الشجرة) **شجرة التحليل المجردة:**

- هي نفس شجرة تحليل قواعد اللغة لكن بدون الأقواس أو begin و end (أي بدون الرموز النهائية غير المفيدة)

-مثلاً من أجل المثال: $(1+2)*3$



شجرة تحليل قواعد اللغة



ملاحظة: كما نعلم فإن لكل رمز نهائي نوع وقيمة وحتى نستطيع بناء شجرة التحليل المجردة يجب أن نجعل للرموز غير النهائية أيضاً قيمة بالإضافة إلى نوعها (مثلاً في LL يصبح كل تابع (التابع حتماً مقابل لرمز غير نهائي) يعيد node ولا يوجد لدينا توابع تعيد void) **مثال:** لنأخذ expression كمثال:

حيث سنكتب إجراءات LL نقوم بحساب هذا التعبير الرياضي وإعطاء ناتجه بدل من الاكتفاء من التأكد من صحته القواعدية

$E \rightarrow T + E$
| T
 $T \rightarrow F * T$
| F

$$F \rightarrow \underline{\text{num}}$$

$$| (E)$$

سنكتب تابعين من توابع LL:

```
int F()
{
    Token t=lex.nextToken();
    Switch(t.type){
        NUM:
            return Integer.parseInt(t.value);
        LPAR:
            temp=E();
            t=lex.nextToken();
            if (t.type==RPAR)
                return temp;
            else
                return -∞;
```

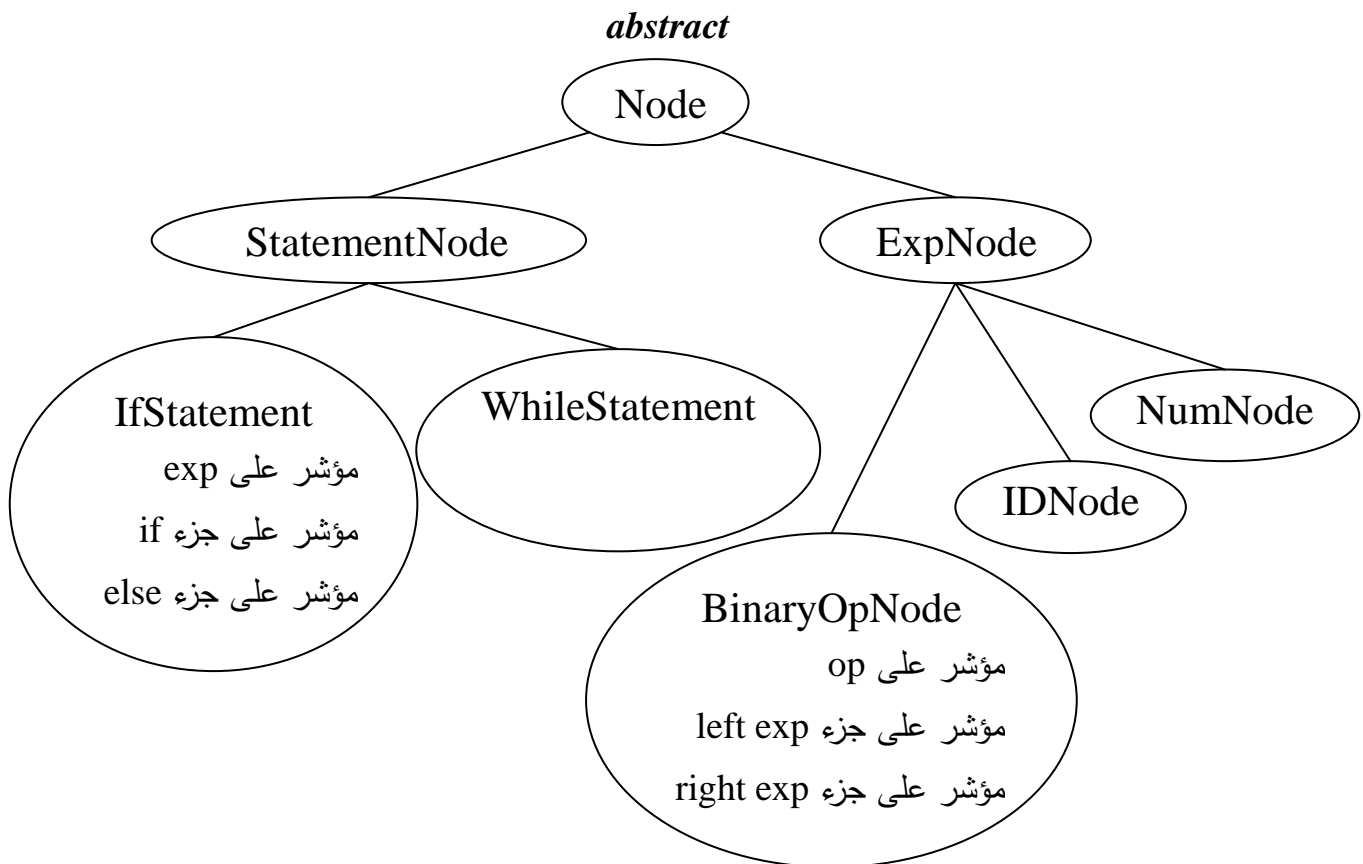
// حتى يعرف المستوى الأعلى بوجود خطأ

```
    }
}
int E()
{
    int temp1=T();
    Token t=lex.nextToken();
    If(t.type!=PLUS)
    {
        lex.unget();
        return temp1;
    }
    else
    {
        int temp2=E();
        return temp1+ temp2;
    }
}
```

ملاحظة: طبعاً يتوجب علينا فور الحصول على temp1 أو temp2 التأكد من أنهما ليسا $-\infty$ ولكننا لم نكتب ذلك الآن لتبسيط الأمور

- لتوليد شجرة التحليل المجردة نعرف البنية node ونجعل كل تابع (التابع مقابل لرمز غير نهائي حتماً) يعيد غرض من node

- يمكننا تعريف هيكلية هرمية من class node ، مثلاً كما يلي:

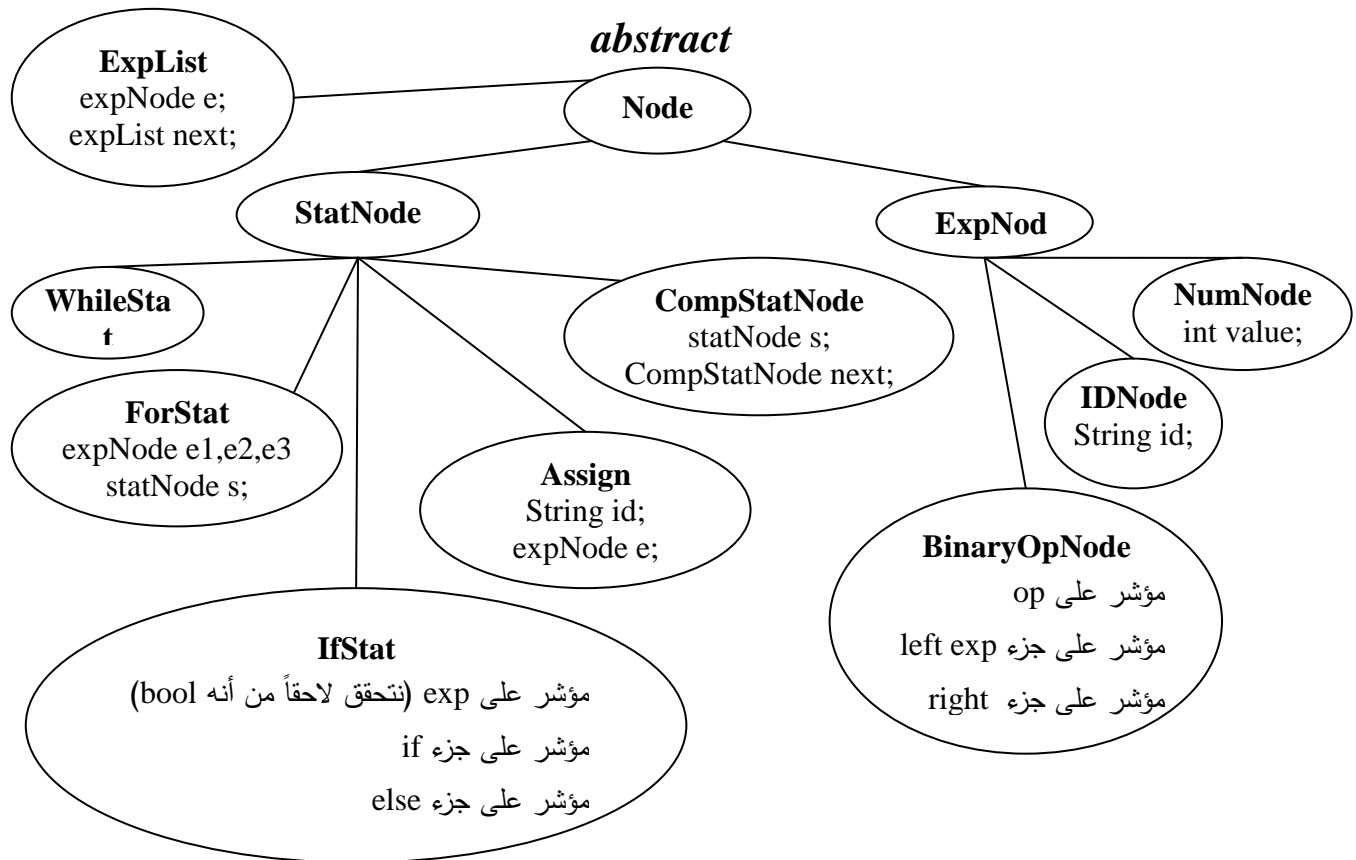


That 's all folks

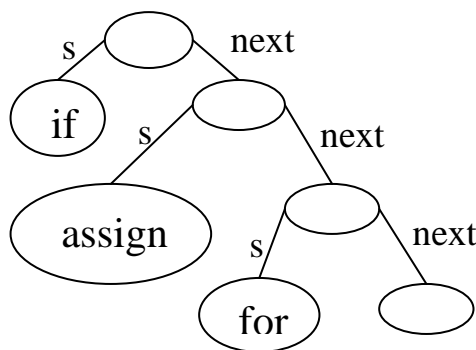
التحليل الدلالي

الاسبوع ٢٠/١١/٢٠٠٦

- ذكرنا في المحاضرة السابقة أنه لتوليد شجرة التحليل المجردة نعرف البنية node ونجعل كل تابع (التابع مقابل لرمز غير نهائي حتماً) يعيد غرض من node



قد نضطر أحياناً لاستخدام compStatNode كما عندما نعرف شيء مشابه لـ block يتضمن عدة statements فنحصل على الشجرة التالية مثلاً:



ملاحظة: نستخدم expList عند استدعاء call لـ function (ليس له value) لأنه يمرر عنصر

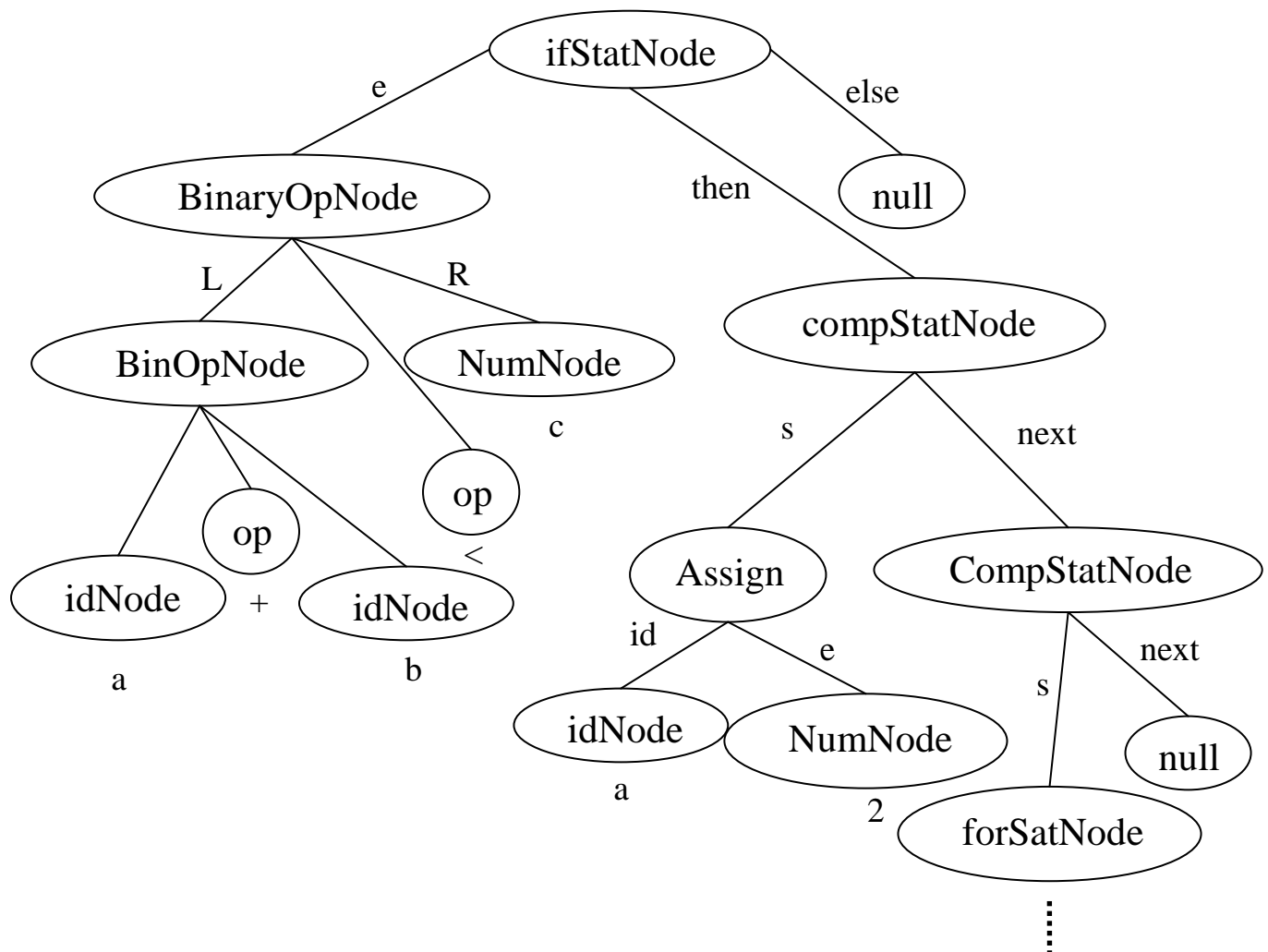
عنصر

مثال شامل:

```

if (a+b < c)
{
    a=2;
    for (i=1;i<=10;i++)
        x=x+1
}

```



توليد الشجرة أثناء التحليل:

مثال exp:

$$\begin{aligned}
 E &\rightarrow T + E \\
 &\quad | \quad T \\
 T &\rightarrow F * T \\
 &\quad | \quad F \\
 F &\rightarrow \underline{\text{num}} \\
 &\quad | \quad \underline{\text{id}} \\
 &\quad | \quad (E)
 \end{aligned}$$

سنكتب الآن بعض الإجراءات التي نستخدم عندما نريد إعادة شجرة التحليل

```

private ExpNode F(){
    Token t=lex.nextToken();
    switch(t.type){
    case ID:
        return new IdNode(t.value);
    case NUM:
        return new NumNode(Integer.parseInt(t.value));
    case LPAR:
        expNode e=E();
        t.lext.nextToken();
        if (t.type == RPAR)
            return e;
        else
            error
    }
}

private ExpNode T(){
    ExpNode e1=F();
    Token t=lex.nextToken();
    if (t.type==STAR){
        ExpNode e2=T();
        return BinOpNode(e1,e2,"*");
    }
    else{
        lex.unget();
    }
}

```

```

    return e1;
  }
}

```

يمكن أن نكتب أيضاً إنشاء الشجرة ضمن برنامج إنشاء parser كما في javacc:

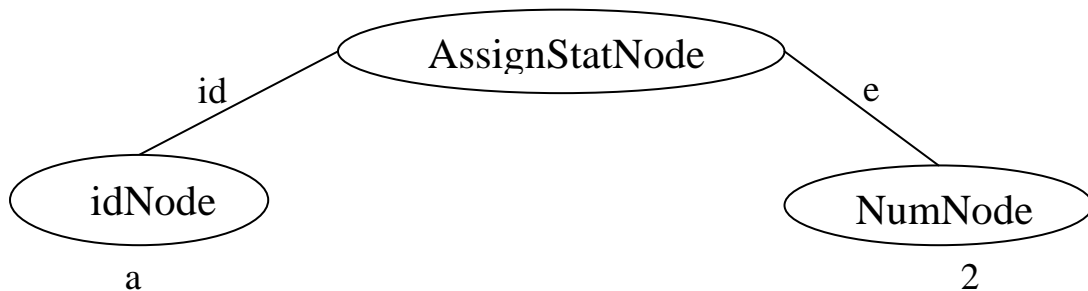
```

S → if (E) S return new ifStatNode(e1,s1,null) // S ⇔ s1 , E ⇔ e1
    | if (E) S else S return new ifStatNode(e1,s1,null) // S ⇔ s1,S ⇔ s2,E ⇔ e1
S → {SL}
SL → S;SL return new statListNode(s,m) // S ⇔ s , SL ⇔ m
    | ε

```

التحليل الدلالي Semantic analysis:

- أي التحقق من صحة البرنامج دلاليًا
- وتطابق الأنماط فمثلاً $a = 2$ ممنوعة في Java إذا كانت a من نوع `bool`
- للقيام بالتحليل الدلالي فإننا نرسل الشجرة السابقة إلى مرحلة التحقق الدلالي
- حيث نبحث في التحقق الدلالي عن a في جدول الرموز ونتأكد من تطابق الأنماط
- كما أننا نقوم بحساب أنماط التعبيرات حيث يعيد التعبير: $a = (\text{int}) b + (\text{float}) c$ النمط الأقوى بين نمطي a و b والذي هو `float`
- لنفرض أننا نريد أن نقوم بالتحقق الدلالي من الشجرة الخاصة بـ $a=2$



للتحقق الدلالي هنا نقوم بما يلي:

- نبحث في جدول الرموز عن المتحول a
- نستدعي التابع `findtype` لمعرفة نمط a
- نختبر التطابق حسب اللغة فمثلاً `int=float` مقبولة في C++ بينما غير مقبولة في java

مثال عن التابع `findtype`:

```
Public string findtype(ExpNode e)
```

```
{
    If (e instanceof NumNode)
        //return depends on the value (real, integer)
    ...
}
```

جدول الرموز **symbol table**:

- من أهم أدوات التحليل الدلالي يحتوي على

- تعريف المتحولات: متحول ← نمط
- تعريف التتابع: متحول ← نمط القيمة المعادة، قائمة الوسطاء
- تعريف class: متحول ← قائمة الخصائص، قائمة التتابع، قائمة الصفوف الداخلية (inner class)

- غالباً يكون لكل class جدول رموز خاص به

- عندما يكون compiler عبارة عن مسحة واحدة نقوم ببنائه فوراً ويكون لدينا التحليل الدلالي وتوليد الـ code مدموجة مع parser

- بينما في حال وجود أكثر من مسحة للـ compiler عندها نقوم بإنشاء جدول الرموز العام في المسحة الأولى (المسحة الخاصة بالتعرف على classes و fields و functions الخاصة بهم) وفي المسحة الثانية نقوم بالإضافة على جدول الرموز والحذف منه حسب المكان الذي نحن فيه أثناء المسح (قد نضطر عندها لأن نخزن في جدول الرموز statements أو بعض المعلومات عنها على الأقل)

That's all folks



lectures_team@hotmail.com

الاسم: ٤/١٢/٢٠٠٦

جدول الرموز Symbol table

بنية جدول الرموز:

- يدعم جدول الرموز إضافة رموز وحذفها والبحث عنها (بسرعة)

- يستعمل عادة جدول التقطيع hash table لتخزين الرموز

أنواع الرموز:

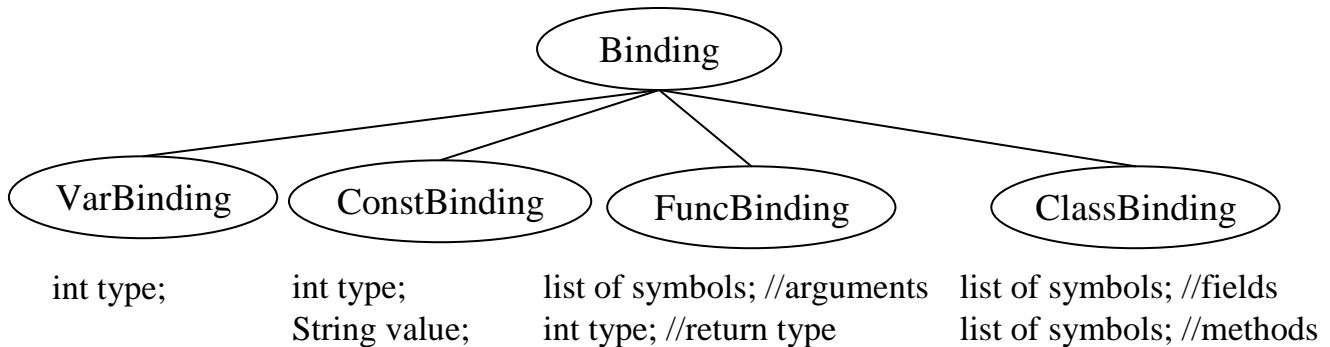
- متحولات
- ثوابت
- توابع
- صفوف

• نمط (type) كما هو typedef في C++ أو type في pascal

ملاحظة: يهمننا من أجل المتحولات أن نخزن نوعها وقد نخزن value من أجل عمليات

optimization (بينما في interpreter يجب علينا حتماً تخزين القيمة مع المتحول لأننا ننفذ فوراً)

- كما قد يحتوي جدول الرموز على labels إذا كانت اللغة تدعم ذلك



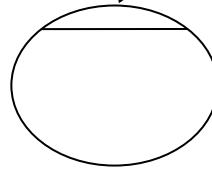
- يتكون كل رمز symbol في جدول الرموز من string يمثل اسم المتحول أو التابع أو الصف

أو ... بالإضافة إلى object من نوع Binding يحتوي على معلومات عن هذا الـ symbol

- ويكون عنصر الجدول على الشكل: (مثلاً متحول)

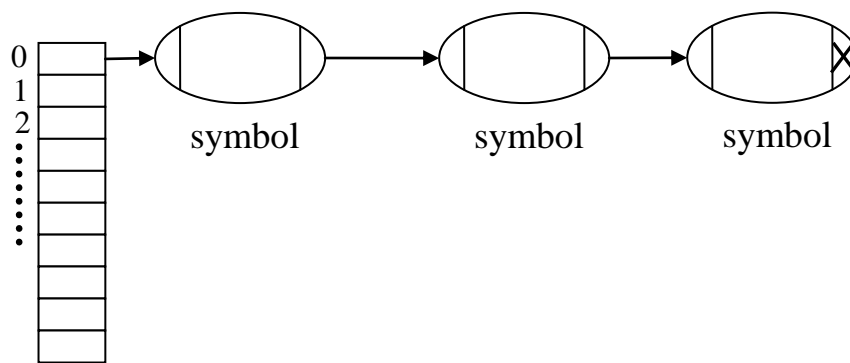
Symbol

"a"



VarBinding

جدول التقطيع :HashTable



-نستخدم تابع تقطيع لربط string الخاص بال symbol مع رقم index لا symbol في جدول التقطيع، ثم نضيف دائماً في بداية List المقابلة لهذا ال index

-لاحظ أن الإضافة في بداية ال list تقدم ميزة تغليف المتحولات العامة بالمتحولات المحلية إذا كان لها نفس الاسم لأن المتحول المحلي ستنتم إضافته إلى ال list بعد المتحول العام وبالتالي فإن البحث عن المتحول سيقودنا لمسح list من بدايتها مما سيسبب ظهور المتحول المحلي وليس المتحول العام

:Code

```
class Symbol{
    String key;
    Binding binding;
    Symbol next;
    public symbol(String k, binding b, symbol n)
```

```

    {
        key=k;
        binding=b;
        next=n;
    }
} //class Symbol

```

```

class HashT {
    final int SIZE = 256;
    Symbol table[] = new Symbol[SIZE];

    private int hash(String s) {
        int h=0;
        for(int i=0; i<s.length(); i++)
            h=h*65599+s.charAt(i);
        return h;
    }

    void insert(String s, Binding b) {
        int index=hash(s)%SIZE
        table[index]=new Symbol(s,b,table[index]);
    }

    Object lookup(String s) {
        int index=hash(s)%SIZE
        for (Binding b = table[index]; b!=null; b=b.next)
            if (s.equals(b.key)) return b.binding;
        return null;
    }

    void pop(String s) {
        int index=hash(s)%SIZE
        table[index]=table[index].next;
    }
} //class HashT
class SymbolTable

```

```

{
    HashT tab=new HashT();
    Stack st=new Stack(); //stack of string
    void insert(String key, Binding b)
    {
        tab.insert(key,b);
        st.push(key);
    }
    Binding lookup (String key)
    {
        return tab.lookup(key);
    }
    void startScope()
    {
        st.push(null);
    }
    void exitScope()
    {
        String s=st.pop();
        while (s!=null)
        {
            table.pop(s);
            s=st.pop();
        }
    }
}

} // class SymbolTable

```

- هذا الكلام يكفي تماماً للغات ذات المسحة الواحدة

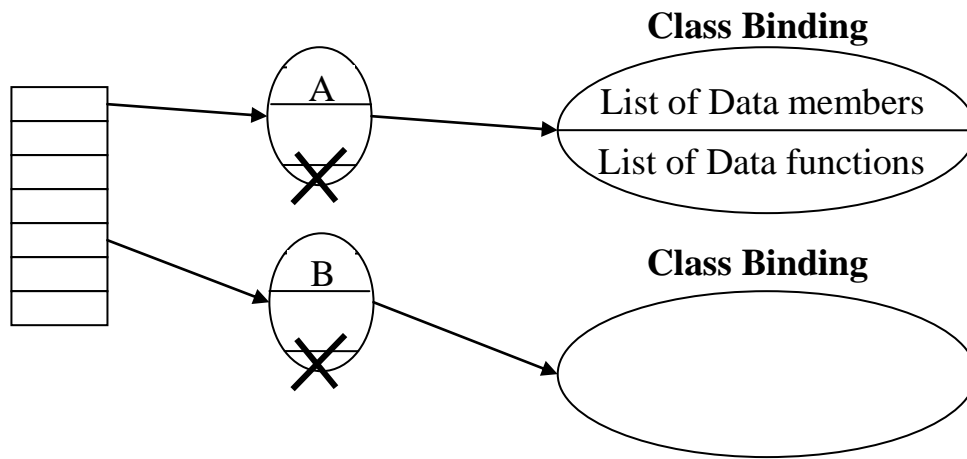
- قد نقوم بإيجاد symbol table لكل function بالإضافة لـ symbol table عام يضم المتحولات العامة وكل classes العامة

جدول الرموز في اللغات الغرضية التوجه:

في هذه اللغات يقوم المترجم بإجراء مسحتين للمصدر:

- الأولى: لا تدخل لتعليمات التتابع بل تولد جدول الرموز (الأعلى) يحوي فقط رموز من نوع classbinding يمكنها أيضاً أن تقوم بالتحقق القواعدي فقط من التتابع (نخزن أشجار تحليلها في جدول الرموز)

- الثانية: التحقق الدلالي من التتابع



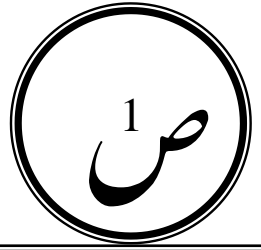
That 's all folks



lectures_team@hotmail.com

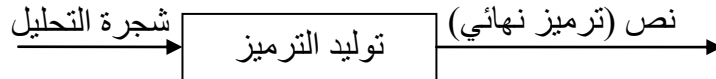


الترجمات



الاسبوع ١١/١٢/٢٠٠٦

توليد الترميز النهائي



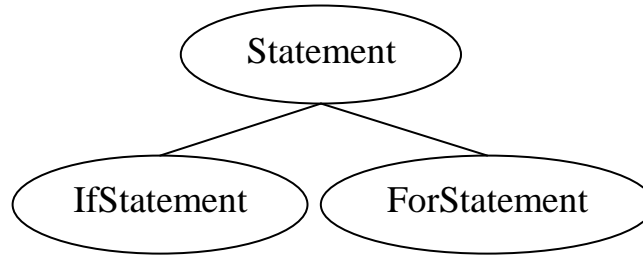
سنفترض تعليمات لغة الآلة التالية:

\$t0, \$t1, ...	(سجلات غير منتهية مبدئياً)
lw \$t0, v(\$t1)	
sw \$t0, v(\$t1)	
add \$t0, \$t1, \$t2	(\$t0=\$t1+\$t2)
sub \$t0, \$t1, \$t2	(\$t0=\$t1-\$t2)
mul \$t0, \$t1, \$t2	(\$t0=\$t1*\$t2)
div \$t0, \$t1, \$t2	(\$t0=\$t1/\$t2)
ldi \$t0, v	load immediately (put v in register \$t0)
J label	jump to label
beq \$t0, \$t1, label	branch if equal
bne \$t0, \$t1, label	branch if not equal
blt \$t0, \$t1, label	

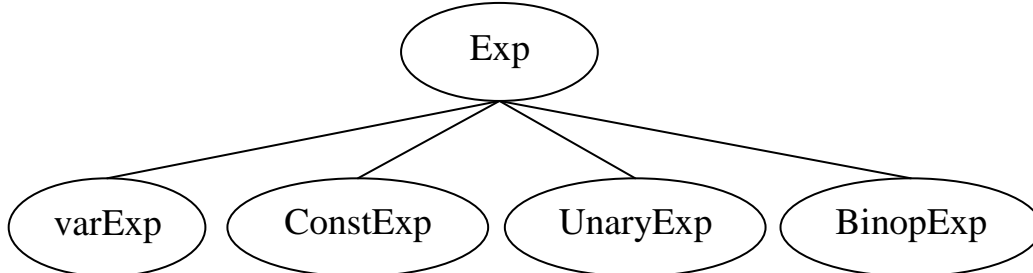
- لتوليد الترميز النهائي فإننا نستفيد من شجرة التحليل التي قمنا بتوليدها

- فنضع في كل عقدة تابع Generate الذي يكتب الترميز النهائي لهذه العقدة ويستدعي generate الخاصة بأبناء هذه العقدة

Abstract Generate



Abstract Generate



-مثلاً في if يتم استدعاء generate الموجودة في قسم condition وكذلك في قسم if وكذلك generate الموجودة في قسم else وحتى نستطيع تنفيذ if لابد أن يعيد كل generate اسم السجل الذي سيضع فيه القيمة النهائية الناتجة عن تنفيذه (يتضح ذلك في مثال exp بشكل أكثر وضوحاً فلكتابة الترميز النهائي لـ BinopExp لابد من أن نكتب الترميز النهائي لـ exp اليميني وكذلك exp اليساري ثم نعرف أين وضع كل منهما ناتج تنفيذه حتى نستطيع أن نقوم بالعملية الثنائية مثل الجمع عليهما)

-لذلك نجعل generate تعيد شيء من نمط code يحتوي على الترميز النهائي وكذلك اسم السجل الذي ستضع فيه الجواب:

```

class code{
    public String code;
    public String reg;
}
  
```

سنفرض وجود إجرائية تدعى nextRegister تقوم بإعطائنا السجل الفارغ الذي يمكن أن نستخدمه للتخزين، في حالتنا هنا وبما أن عدد السجلات غير منتهى فقد تكون هذه الإجرائية عبارة عن عداد يعطينا أرقام متتالية من السجلات ولكن في الحقيقة لابد من نظام إدارة سجلات بحيث نقتصد قدر الإمكان باستخدامها مع المحافظة الدائمة على إمكانية معرفة السجلات الفارغة والممتلئة

أمثلة:

تعريف generate في :constExp

```

class ConstExp extends Exp
{
    private int val;
    //and all necessary fields and methods
    public Code Generate()
    {
        Code c=new Code();
        String r=nextRegister();
        c.code="ldi" + r + "," + val;
        c.reg=r;
        return c;
    }
}

```

تعريف generate في :BinopExp

```

class BinopExp extends Exp
{
    private Exp left,right;
    private String op;
    public code generate()
    {
        Code c1,c2;
        c1=left.generate();
        c2=right.generate();
        Code c3=new Code();
        String r=nextRegister();
        String r;
        switch (op)
        {
            case "+": v="add"; break;
            case "-": v="sub"; break;
            //add all cases of op
        }
    }
}

```

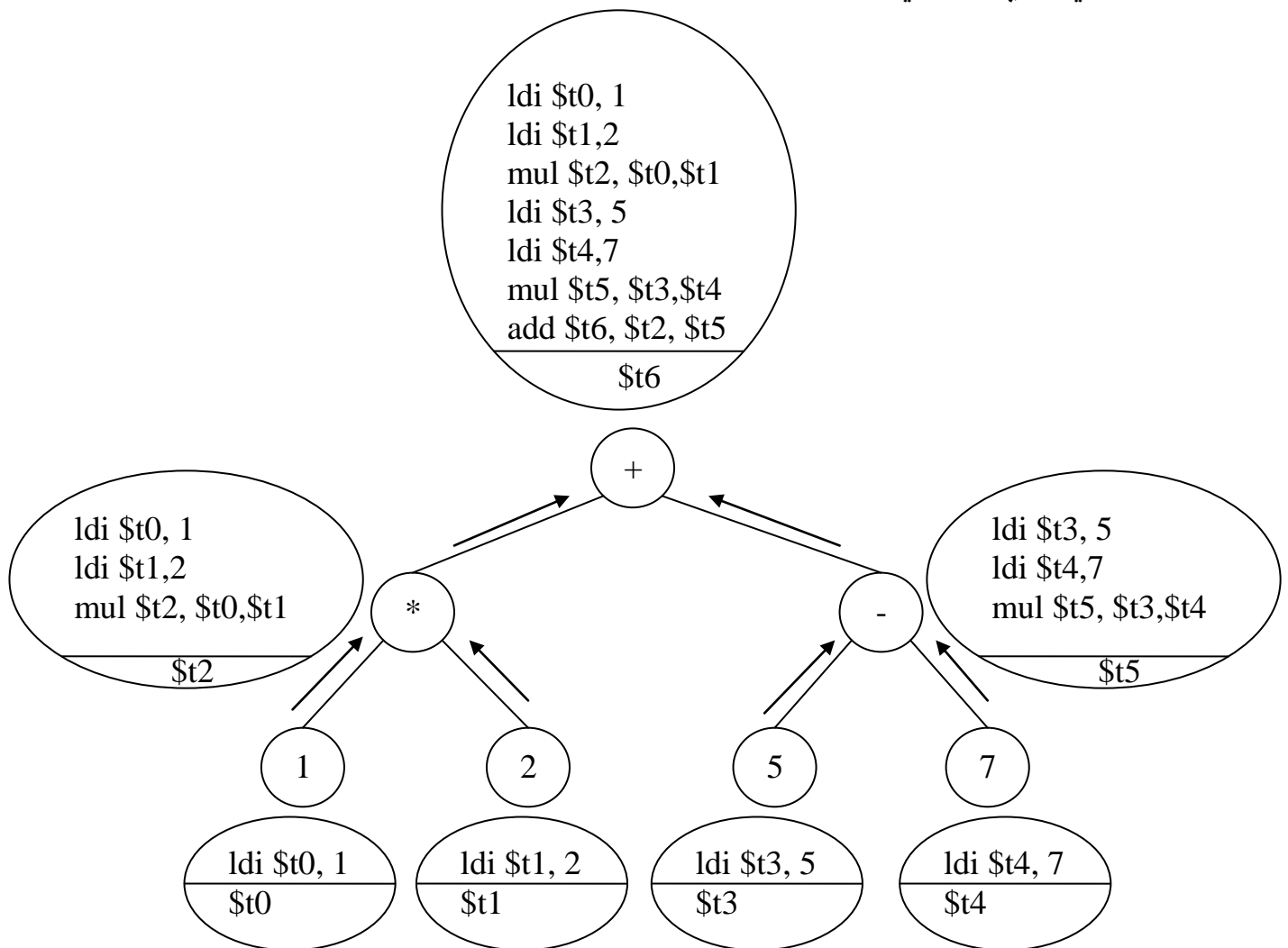
```

c3.code=c1.code+"\n"+c2.code+"\n"+v+r+", "+c1.reg + ", " + c2.reg;
c3.reg=v;
return c3;
}
}

```

ملاحظة: في المثال السابق نحن نسرف باستخدام السجلات فيمكننا ببساطة أن نتخلى عن c3 ونضع الناتج النهائي في c1 أو c2 لأننا لم نعد نحتاج للقيمة الموجودة في كل منهما

لنأخذ المثال التالي الذي يبين في شجرة التحليل ماذا سيعيد generate من أجل كل عقدة فيها



التعليمات:

-مثلاً if ليس لها value وكذلك for بينما = لها value

-نفترض هنا أن كل الـ statements ليس لها value كما في pascal

تعريف generate في Ifstatement:

class IfStatement extends statement

```
{
    private Exp e;
    private Statement then ,else;
    //and all necessary fields and methods
    public Code generate()
    {
        Code c1,c2,v3,c;
        c1=e.generate();
        c2=then.generate();
        c3=else.genereate();
        String l1,l2;
        l1=nextlabel();
        l2=nextlabel();
        c.code=c1.code+"\n"+"beq"+c1.reg+"$,zero,"+l1+"\n"
            +c2.code()+"\n"+"J"+l2+"\n"+l1+": "+"n"+c3.code+"\n"+l2+": ";
        return c;
    }
}
```

تعتمد آلية if على الشكل التالي:

```
{c1} //condition
beq c1.reg, $zero, L1
{c2} //then
J L2
L1:
{c3} //else
L2:
```

That 's all folks